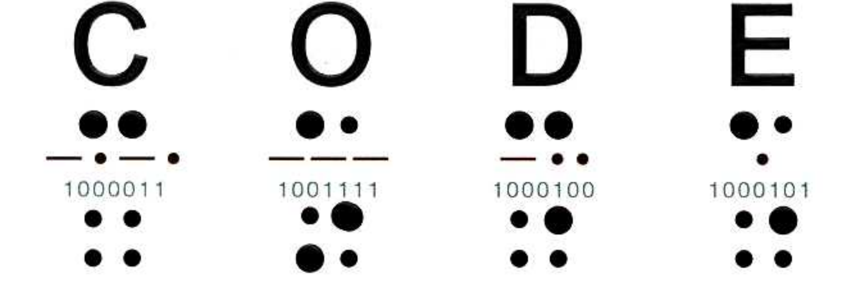


Microsoft

永不褪色的计算机科学经典著作



编 码

隐匿在计算机软硬件背后的语言

Code:The Hidden Language of Computer Hardware and Software

[美] Charles Petzold 著

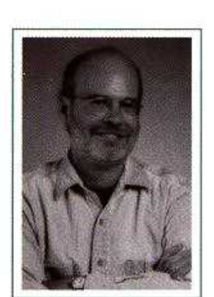
左飞 薛佟佟 译





从零开始构建一台计算机，原来一切如此奇妙！

关于作者：



Charles Petzold 是Windows编程界的一位大师，当今世界顶级技术作家。1994年5月，Petzold作为仅有的七个人之一（并且是唯一的作家）被《视窗杂志》和Microsoft公司联合授予Windows Pioneer奖，以表彰他对Microsoft Windows所做出的贡献。

Petzold从1984年开始编写个人计算机程序，从1985年开始编写Microsoft Windows程序。1986年他在Microsoft Systems Journal的12月号上发表了第一篇关于Windows程序设计的论文。

从1986年到1995年，Petzold为PC Magazine撰写专栏文章，向读者介绍Windows和OS/2程序设计等方面的知识。直到今天他依然保持着Windows GDI程序设计首席技术作家的地位。其大作Programming Windows（Windows程序设计）是尽人皆知的Windows编程经典，曾深深地影响过一代程序员，该书目前已出至第5版。

上架建议：计算机>编程

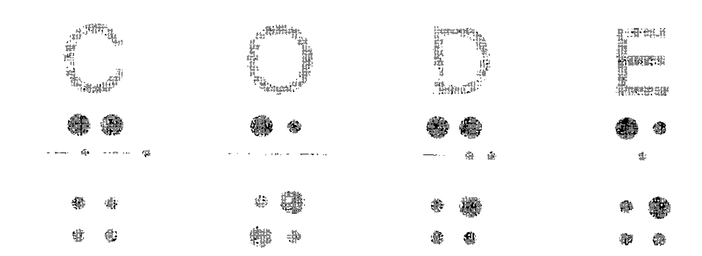
Microsoft®



责任编辑：许　艳
责任美编：李　玲



定价：59.00元



编码

隐匿在计算机软硬件背后的语言

Code:The Hidden Language of Computer Hardware and Software

[美] Charles Petzold 著

左飞 薛佟佟 译

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

这是一本讲述计算机工作原理的书。不过，你千万不要因为“工作原理”之类的字眼就武断地认为，它是晦涩而难懂的。作者用丰富的想象和清晰的笔墨将看似繁杂的理论阐述得通俗易懂，你丝毫不会感到枯燥和生硬。更重要的是，你会因此而获得对计算机工作原理较深刻的理解。这种理解不是抽象层面上的，而是具有一定深度的，这种深度甚至不逊于“电气工程师”和“程序员”的理解。

不管你是计算机高手，还是对这个神奇的机器充满敬畏之心的菜鸟，都不妨翻阅一下本书，读一读大师的经典作品，必然会有收获。

版权贸易合同登记号 图字：01-2009-3031

图书在版编目（CIP）数据

编码：隐匿在计算机软硬件背后的语言 /（美）佩措尔德（Petzold,C.）著；左飞，薛佟佟译. 北京：电子工业出版社，2012.10

书名原文：Code: The Hidden Language of Computer Hardware and Software

ISBN 978-7-121-18118-4

Ⅰ. ①编… Ⅱ. ①佩… ②左… ③薛… Ⅲ. ①计算机科学－理论 Ⅳ. ①TP3-0

中国版本图书馆 CIP 数据核字(2012)第 204356 号

策划编辑：刘　皎
责任编辑：许　艳
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：720×1000　1/16　印张：27.25　字数：442 千字
印　　次：2012 年 10 月第 1 次印刷
定　　价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

编码（biān mǎ）

3. a. 一种在信息传输过程中用来表述字母或数字的信号系统。

b. 由被赋予了一定主观意义的符号、字母以及单词所组成的系统，该系统可用于传输需要保密的或简短的信息。

4. 一种由若干符号和规则组成的系统，用来向计算机表述指令。

——摘自《美国传统英语词典》

code (kod) ...

3. a. A system of signals used to represent letters or numbers in transmitting messages.

b. A system of symbols, letters, or words given certain arbitrary meanings, used for transmitting messages requiring secrecy or brevity.

4. A system of symbols and rules used to represent instructions to a computer…

— The American Heritage Dictionary of the English Language

推荐序

《道德经》有云：大方无隅，大象无形，也就是老子所说“道”的至高境界。世界上最恢宏、壮丽的气派和境界，往往并不拘泥于一定的事物和格局，而是表现出“气象万千”的面貌和场景，科学知识的论述也是如此。科学不一定等同于晦涩和深奥，最复杂的理论也可用最简单的方式描述。这就是普遍存在于现实世界的辩证的、朴素的唯物主义精神。

我想 Charles Petzold 就是这样一位智者。作为 Windows 编程界开创者之一和世界顶级的技术专家，他在 Windows 编程方面著述丰富，在基础计算机理论书籍方面也取得了卓尔不凡的成就。他善用丰富的想象和清晰的笔墨将看似繁杂的理论描述得生动、简单，使读者不会感到枯燥和生硬。尤其这本畅销多年、经久不衰的《编码——隐匿在计算机软硬件背后的语言》(*Code：The Hidden Language of Computer Hardware and Software*)更是化腐朽为神奇，改晦涩为通俗之杰作。即使读者不是计算机专业的人士，在 Petzold 笔墨的引导之下，也能够轻松阅读该书。书中使用像电线、电灯泡、触发器等这些非常常见的零件，拼拼接接，最后令人惊异地建造了一台电子计算机。读者在阅读过程中，循序渐进地学习有关计算机系统基础和构造的知识，作者的奇思妙想和妙趣横生的讲述方式不得不令人赞叹。

透过现象进而探索本质可以发现新事物；将复杂的事物简单化，可以发现解决问题的新方法。作者将那些看似复杂的理论转换成我们熟悉的诸多景象，使读者不由自主地跟随他的思路打造这台只有电线、电灯泡和触发器组构成的庞大的计算机系统。

我阅读了本书的译稿，并非常欣喜地将其推荐给每一位读者，我相信读者都能获益良多。本书的译者做了一件非常有意义的事情，将一部优秀的作品介绍给读者。他们严谨的工作将本书诠释得非常出色，使之在不丧失原作风采的基础上更贴合中国读者的习惯。

最后，我还是想将《道德经》上的两句话来送给各位读者——"为学日益，为道日损，损之又损，以至于无为，无为而无不为"。衷心希望各位读者有朝一日都能够在学问上到达无为而治的境界。

丁秋林

2009.11.28.

丁秋林教授，博士生导师，南京航空航天大学信息科学与技术学院和机电工程学院资深教授。英国皇家航空学会高级会员、英国剑桥世界名人录传记人物（IBC Cambridge the 28th Edition）、美国世界名人录传记人物（Who's Who in the World the 16th Edition）、国际高级制造技术杂志编辑部成员（Springer）。

译者序

在想到要给《编码——隐匿在计算机软硬件背后的语言》(*Code: The Hidden Language of Computer Hardware and Software*)写一篇序言的时候，内心就不由得激动起来。能够翻译这部经典之作实属吾辈之荣幸。早在一年多之前，电子工业出版社的编辑跟我提到这本书的翻译之事时，我便有颇大的兴趣。一是由于该书作者 Charles Petzold 的鼎鼎大名，二是由于该部作品本身内容之不凡。

作为 Windows 编程界的一位传奇人物，Charles Petzold 早在 1994 年就作为仅有的七人之一（并且是唯一的作家）被《视窗杂志》和 Microsoft 公司授予 Windows Pioneer 奖。而作为一名世界顶级的技术作家，他的大作 *Programming Windows* 则是尽人皆知的 Win32 API 编程经典，是学习 Windows 编程的必读之作。令我们欣喜的是，除了 Windows 程序设计方面的著作，Petzold 还撰写了这部讲述计算机工作原理的书籍，而且同样堪称经典。

本书的翻译过程漫长而又愉快，书中的精彩内容不断给我们以惊喜。Petzold 采用的叙述方式奇特却又如此自然，他首先从历史的角度审视了计算机技术的发展脉络，继而自底向上地将计算机体系结构方面的知识娓娓道来。在 Petzold 的笔触下，导线、灯泡以及触发器等简单部件，通过精妙的组装，竟然构建出了一台电子计算机，令我们在翻译的过程中不禁有动手一试的冲动，但总是又被他精彩的笔触所吸引而欲罢不能。Petzold 在这本技术书籍里旁征博引，从美国当红的影评人到百年之前的布莱叶盲文，从史密斯·索尼亚的滑尺到冯·诺依曼的构想，从石破天惊的论文《思维之际》到图形化革命，看似信手拈来，然则处处别出心裁。

这些被有机结合起来的零星知识，仿佛又把我带回了大学的课堂。所涉及的问题包括计算机体系结构、汇编语言及数字电子技术等等。那些已经久远而又模糊的知识骤然间变得异常清晰而且格外生动。这些知识原本只会出现在高校的计算机专业课堂里，令人惊奇的是，尽管遍布于世界各地的本书读者往往有着各不相同的专业背景，而这却丝毫不影响本书内容所带给他们的震撼！

所谓经典，就是历尽岁月的洗涤，依然毫不褪色，然其光彩却历久弥新。我想本书被奉为经典之作当之无愧。

为了能够将该书的精彩原原本本地呈现给广大中国读者，并在保证质量的前提下力争让该书早日与广大读者见面，本书的译者始终秉持着谨慎严肃的态度，对许多词句都推敲再三，力求精益求精。在此期间，我们得到了许多师长和友人的支持与帮助，感激之情溢于言表。特别是南京航空航天大学丁秋林教授，他对我们的翻译工作给予了极大的支持，并欣然为本书提笔作序，在此我们对丁教授表示诚挚的谢意。

参与本书翻译工作的人员还有初甲林、刘悦、高新、高阳、邓明强和赵德志。此外赵学健、欧阳键两位博士也参与了部分章节的审校工作，在此也向他们一并表示感谢。

在本书即将付梓之际，除了激动喜悦之外，译者又多了一份惶恐，尽管我们几经校对，并在翻译过程中始终以谨慎求实的态度对一些技术细节和翻译方式反复讨论，力求表意准确。然而限于时间仓促，纰漏和欠缺之处在所难免，言语之中有失偏颇之处，还望读者不吝赐教和批评。联系信箱：beckham@vip.163.com。

薛佟佟

2009 年秋于南京

作者序

在真正开始撰写这本书之前，纷繁的思绪在我的脑海中已经萦绕了近十年之久。但从着手写作，一直到出版，我不断反复地问着自己一个问题：这本书是讲什么的？

关于这个问题我总是找不到合适的答案。我想说："这本书将带领大家通过一段信息技术革命的旅程重新感触现代计算机技术。"语毕，我内心纠结，这个回答真的将这本书的内容完整诠释出来了吗？

最后我不得不承认："这本书其实是讲述计算机如何工作的。"

我之所以有些底气不足，那是因为我已经猜到了某些读者的反应："嗨，这种书我早就读过了。"但我会立刻告诉他："不，这种书你并没有读过。"对于这点，我坚信不疑。因为这本书并不是讲"计算机——如何——工作"。书中并没有用一张又一张描述磁盘的插图，用各种箭头解释数据是如何被输送到计算机的，书里面也没有装载着一节又一节"0，1 数据"的"火车"图片。明喻与暗喻是文学描述中精妙的辅助手段，但它们常常掩盖了科学技术的真正光芒。

我还听过另一个说法："人们不愿意了解计算机如何工作。"对于这句话我毫无保留地赞同，因为就我个人来说，也是在不经意之间才体会到理解事物如何运作这一过程是多么有趣。但是这并不意味着我对世上一切事物的运作机制都感兴趣，都有所了解，因此我绝对不会在任何场合向大家解释我房间里冰箱的工作原理。

然而与此同时，经常有人向我问起一些涉及计算机内部运行机理的问题。一个常见

的例子就是："'存储器（storage）'与'内存（memory）'有何区别？"

对于计算机用户而言，这个问题要是搞不清楚，的确"寸步难行"，可以看到，市场上考察个人计算机的存储性能，最主要的就是这两个概念。即便对于最初级的计算机用户来说，他们也一定需要了解到底多少"兆字节"或多少"吉字节"的存储器才能应对运行在其上的程序。如果进一步去思考，这些初级用户或许更加想了解计算机中的"文件"是什么概念，甚至连带这些文件如何从存储器加载进内存，又如何从内存存储到存储器，他们也非常期望学习这些知识。

像这类"存储器-内存"问题的解答大都使用类比法："内存就好比你的办公桌，而存储器就好比你的文件柜。"就问题本身而言，它的确给出了满意的回答。但我对此答案并不满意，主要原因在于这个答案将计算机的体系结构与办公室的结构等同起来，但是内存与存储器的区别其实是在逻辑层面上的，它体现着计算机体系结构的实际需求与存储器客观性能之间的矛盾，简单地说就是我们找不到一种同时具备这两种存储器所有优点的存储媒介，这些优点就包括存储速度块、存储容量大、非易失性等等。今天的计算机都采用"冯·诺依曼体系结构"——五十年来它一直是计算机体系结构的主导，而内存与存储器的区分也正是由于这种体系结构的不足所导致的。

还有一些计算机用户问我这样一个问题："为什么 Macintosh 环境下的程序不能在 Windows 下运行？"我想立刻回答这个问题，但在我刚刚张开嘴的时候就立刻意识到，这个问题涉及太多的技术细节，要想彻底搞清楚，那提问的这位朋友也一定非准备和我来一次"茶话会"不可。

我希望这本书能够成为大家理解这些问题的"助手"，这种理解我希望不是抽象层面上的，而是具有一定深度的，这种深度甚至不逊于"电气工程师"和"程序员"的理解。我同时也非常希望大家能够理解：计算机是二十世纪技术领域的"登峰造极之作"，它是一种值得欣赏、具有"美"学文化底蕴的人类伟大成果，这种"美"不需要明喻与暗喻的额外修饰。

计算机拥有与生俱来的层次化体系结构，这种结构的底层是晶体管，其顶层则是计算机显示器上所呈现的信息。自底向上分析该结构的每一层——这也是本书的编写结构——其实这一切并没有人们想象中那么难。当然，现代计算机的内部结构不断推陈出新，但其本质上仍然是一些常见且简捷的操作集合。

尽管今天的计算机比起 25 年前，以及 50 年前的都复杂许多，但它们在本质上是完全一致的。学习技术发展史的重要意义正在于此：追溯的历史越久远，技术的脉络就变得越清晰。因此，我们需要做的就是确定某些关键的历史阶段，在这些阶段，技术最天然、最本质的一面将清晰可见。

在这本书中，我回溯了自己所能找得到资料的计算机发展史。令我自己也感到惊讶的是，有时竟然一直追溯到 19 世纪，甚至使用了早期的电报设备来演示计算机是如何构建的。至少从理论角度来看，本书的前 17 章中提到过的所有设备，都可以利用已经存在了一个世纪的简单电子器件来构造。

这些古董级技术的使用令本书蒙上了一层怀旧的面纱。我要强调的是 *Code* 是这样一本书，它永远不会被命名为诸如 *The Faster New Faster Thing* 或 *Business @ the Speed of a Digital Nervous System*。在这本书中，“bit” 定义在第 70 页，“byte” 的定义则出现于第 186 页。晶体管直到第 146 页才被介绍，而且只是顺便提及而已。

在对计算机工作原理介绍方面，本书将会一直深入到本质（例如，只有少数书会去介绍计算机处理器的实际工作机制），但整本书的节奏是相对缓和的。在保证内容深度的基础上，我尽量使读者在学习的旅程中保持轻松愉悦的心情。

我最后还是要说，书里面没有那种装载着一节又一节 “0，1 数据” 的 “火车” 图片。

Charles Petzold

2000 年 8 月 16 日

目　录

14 反馈与触发器 / 160

想象一下，如果你没有了记忆力，该如何去数数？我们不记得刚刚数过的数，当然也就无法确定下一个数是什么！同理，一个能计数的电路必定需要触发器。本章要介绍的就是各种触发器。

15 字节与十六进制 / 186

在前面的章节中，加法器、锁存器以及数据选择器的输入和输出形式都是8位的数据流，也即数据路径的位宽为8，为什么要定义为8位呢？为什么不是6位、7位、9位或10位？本章就要解释其中的缘由。

16 存储器组织 / 197

每天清晨，我们将自己从沉睡中唤醒，这时大脑的空白会很快被记忆填满。我们立刻会意识到自己身在何方，最近做了些什么事情，有什么计划打算。有的事情我们很快就能想起来，但有时并非如此。我们可以借助许多工具来记录信息，比如笔和纸、磁带，当然现在还可以使用存储器。

17 自动操作 / 215

人类的本性中带有一些懒惰的特质。我们总是抵触繁重的工作，对枯燥的、重复性的工作深恶痛绝。所以，当你必须用前面搭建的加法器计算100个数，甚至更多个数的加法时，有一种念头就会不可遏制地从脑子里冒出来：怎样让加法器自动地完成数据输入和计算呢？办法肯定是有的，那就是编写程序。

18 从算盘到芯片 / 252

算盘、滑尺、纳皮尔骨架、差分机、解析机、继电器、电子管、晶体管、芯片、计算机；甘特、帕斯卡、莱布尼兹、杰奎德、巴贝芝、图灵、冯·诺依曼、香农；IBM、贝尔实验室……你觉得应接不暇了吗？把这些你或熟悉或生疏的名词和名字串起来，就是人类的计算工具发展史。让时光倒流，去看看那些精巧的工具，感受天才们的巧思吧！

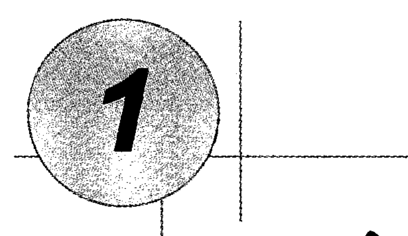
内存里，所有的数都表示为二进制形式。通过前面的学习，我们知道 2 用二进制可以表示为 10_2，可是 2.75 用二进制怎样表示呢？这就是本章的主题。

24 高级语言与低级语言 / 381

第 22 章介绍了如何编写一段简单的程序，让我们可以利用键盘将十六进制机器码输入计算机，以及通过视频显示设备来检查这些代码。但是使用机器码编写程序就如同用牙签吃东西，伸出手臂费半天劲刺向食物，但每次都只取到小小的一块，用这种低级语言编写程序既费力又费时，有悖于我们发明计算机的初衷。不过，人们想出了一种效率更高的编程方法——使用高级语言。

25 图形化革命 / 398

回顾历史，从第一台继电器计算器到现在为止，六七十年过去了，计算机的处理速度飞速增长。不过要充分利用计算机日益增长的运算和处理能力，就必须不断改进计算机系统中的用户接口（User Interface），因为它是人机交互的轴心。图形化革命来了！



至亲密友

你今年 10 岁，你最好的朋友就住在街对过。事实上，你们各自卧室的窗户正好彼此相对。每当夜幕降临，父母就如同往常一样，早早地催促你该上床睡觉了，但是你和你的朋友还想交流想法，交换见闻，分享各自的秘密，或者扯扯闲话，开开玩笑，聊聊梦想。这本无可厚非。无论怎样，渴望交流本来就是人类最主要的天性之一。

当卧室里的灯依然亮着的时候，你可以和朋友互相挥手，使用各种手势或简单的肢体语言，来表达一两个想法。但是，要表达复杂的想法可能就比较困难了。而且一旦父母宣布“关灯”，这种交流似乎也不可能继续下去。

如何交流呢？或许可以打电话？10 岁小孩的房间里会有电话吗？即使有，无论电话在哪里，你们的谈话都有可能被偷听。如果你家里的电脑连接了电话线，它可能帮上忙，而且不会发出声响，但是——等等，电脑也不会在你的房间里。

你和朋友所采用的方法就是使用手电筒。众所周知，手电筒是为了让孩子们能够躲在被子下看书而发明的；在天黑后用手电筒来交流信息似乎也是理想的选择。它们当然是很安静的，并且光线是高度定向的，同时光线也不会渗漏出卧室而引起家人的疑心。

手电筒能用来交谈吗？这当然值得一试。我们在一年级的时候学习怎样在纸上写字

母和单词，因此，把同样的方法运用到手电筒上似乎也是有道理的。只需要站在窗户边，用光来画出字母。对于字母“O”，打开手电筒，在空气中划一个圈，然后关掉手电筒。对于字母“I”，竖着划一下。但是，你很快就会发现，这个方法也不太行得通。当你看着朋友的手电筒在空中圈圈点点时，会发现很难在头脑中组合出那么多复杂的笔画。这些旋转和倾斜的光线都太不准确了。

或许大家都曾经在电影里看到这样的情节，两个水手在海上通过灯的亮灭来互相发送信号。而在另一部电影里，一个间谍转动一面镜子将太阳光反射到另一个房间里，从而向被俘的同伙传递信息。或许那正是解决问题的办法。如此一来，你就可以发明一种简单的技术。在这个方案里，字符表里的每个字符对应一连串的手电筒闪烁。“A”是闪一次，“B”是闪两次，“C”是闪三次，依此类推，“Z”就是闪 26 次。单词 BAD 可以用闪 2 次，闪 1 次，闪 4 次这样的一个组合来表示，而且在字符之间设置的小停顿使这个单词不至于被误认为是闪 7 次的字母“G”。另外，单词之间停顿可以稍长些。

这似乎很有希望，采用这种方案的好处是你不必在空中比划手电筒了，只要对准方向和按开关就行了。但是这种方案也有一个不足，那就是如果你想发送的第一个消息是“How are you? ”，那么你将总计需要让手电筒闪 131 次！而且，这还是忽略了标点符号的，你还没有设计闪多少次来对应一个问号。

但是这已经离答案更近一些了。我们能够肯定的是，在此之前一定有人也遇到过类似的问题，而你解决这个问题的思路也是非常正确的。等到了白天，跑一趟图书馆，查查资料，你发现了一个被称为莫尔斯电码（Morse Code）的伟大发明。这正是你想找的，尽管你现在必须重新学习如何去“写”字母表里的字母。

它们的不同之处在于：在你发明的系统里，字母表里的每个字母就是用一定数目的闪光表示的，闪 1 下为“A”，闪 26 下为“Z”。而在莫尔斯电码里，则有两种闪烁——短闪和长闪。当然这使得莫尔斯电码更加复杂，但是在实际应用里它被证明是更为有效的。句子“How are you?”现在只要闪 32 下（包含一些短闪和一些长闪），而不再是 131 下，而且这其中还包括了一个代表问号的编码。

当问及莫尔斯电码是如何工作的时候，人们并不会谈论“短闪”与“长闪”。相反，他们使用“点（dot）”和“划（dash）”，因为这是在打印纸上显示编码的一个便利方法。在莫尔斯电码里，字母表里的每个字母都与一个点划序列相对应，正如下表所示。

A	•—	J	•———	S	•••
B	—•••	K	—•—	T	—
C	—•—•	L	•—••	U	••—
D	—••	M	——	V	•••—
E	•	N	—•	W	•——
F	••—•	O	———	X	—••—
G	——•	P	•——•	Y	—•——
H	••••	Q	——•—	Z	——••
I	••	R	•—•		

虽然莫尔斯电码和计算机毫无关系，但是，熟悉编码的本质对于深入理解计算机软硬件内部结构以及隐匿在其后的语言将大有裨益。

在这本书里，编码这个词的意思是指一种用来在机器和人之间传递信息的方式。换句话说，编码就是交流。有时候我们认为编码就是指秘密的东西（密码）。但是大部分编码不是这样的。毕竟，大部分编码必须易于理解，因为它们是人类交流的基础。

在《百年孤独》这本书的开篇里，加西亚·马尔克斯回忆了一个时代，那时“这个世界刚刚出现，以至于很多东西缺乏命名，这时就有必要亲自用手指明这些事物”。我们赋予这些东西名字时往往是很随意的。这就好比说为什么猫不被叫做“狗”而狗不被叫做“猫”一样，没有什么理由可言。你可以说英语词汇就是一类编码。

对任何能听见我们的声音并理解我们所说的语言的人来说，我们发出的声音所形成的词语是一种可识别的编码。我们将这个编码称为“口头话语（the spoken word）”或“言辞（speech）”。对于写在纸上（或刻在石头上、木头上，或者在空气中比划）的词，我们还有其他的编码方式。这种编码以手写字符或打印在报纸、杂志以及书本上的字符形式出现。我们叫它“书面语言（the written word）”或“文本（text）”。在许多语言里，语言和文字之间存在着很紧密的联系。例如，英语中的字母和字母组合与它们的发音（或多或少）存在一定的对应性。

对于那些丧失听说能力的的聋哑人而言，人们发明了另一种编码来帮助他们进行面对面的交流。这就是手语。手语通过手和臂膀形成的动作和姿势来传达词语中的单个字符或者整个词语，以及基本的概念。对于那些失明的人，书面语言可以用布莱叶盲文（Braille）来替代。这种文字使用一系列凸起的点来代表字母、字母串以及整个单词。当

话语必须快速抄成文本时，缩写和速记都是很有用的。

我们使用各种不同的编码来为我们自己的交流服务，因为有些编码有时比其他编码更便捷。例如，话语的编码不能存储在纸上，因此，书写的编码就被用来替代话语的编码。如果在黑暗的环境下，而且交流的双方之间有一定的距离，那么通过讲话或者文字来进行秘密的信息交换几乎是不可能的，因此，莫尔斯编码就成了一个方便的选择。如果一种编码可以用在其他编码无法取代的地方，那么它就是一种有用的编码。

我们将会看见，各种类型的编码也用在计算机里来存储和传递数字、声音、音乐、图片和电影。计算机不能直接处理人类的编码，因为计算机无法通过与人类的眼睛、耳朵、嘴巴和手指完全相同的方式来接收人类发出的信息。然而，计算机技术的一个最新趋势，已使得我们的个人计算机能够获取、存储、处理和呈现一切用于与人类沟通的信息，无论视觉信息（文字和图片），还是听觉信息（口语、声音和音乐），或两者的相结合（动画和电影）。所有这些类型的信息都需要它们各自的编码，就像人类说话需要一套器官（嘴和耳朵）而写作和阅读需要另一套（手和眼）一样。

甚至前面所列的莫尔斯编码表，其本身就是一种类型的编码。在这个表中，每个字母由一系列的“点”和“划”来表示。然而实际上我们不能发送“点”和“划”，相反，我们发送与“点”和“划”对应的闪烁光。

当使用手电筒发送莫尔斯编码时，迅速地打开和关闭开关代表一个“点”（快闪），让闪光时间保持得相对长一些代表“划”（慢速闪光）。例如在发送字母 A 时，首先以非常快的速度打开并关闭手电，然后再以稍慢的速度进行一次。在发送下一个字符前，需要暂停一会。在此约定，一个“划”的时长是“点”的 3 倍。例如，如果一个“点”的时长是 1 秒钟，那么一个“划”的时长就应当是 3 秒钟。（在现实中，莫尔斯编码的传输速度远比这要快得多）。接收者看到一个短促的闪烁和一个拖长的闪光后，就知道这是一个 A 了。

在莫尔斯编码中，“点”和“划”之间的停顿是至关重要的。例如，当发送一个字母 A 时，在发送的“点”和“划”之间，手电筒要关闭一段时间，这相当于一个点的闪烁时长（如果“点”的时长是 1 秒钟，那么“点”和“划”之间的停顿也应该是 1 秒钟）。对于同一个单词中的字母，则通过较长的停顿来分隔，这大约相当于一个“划”的时长（或者说是 3 秒钟，如果一个“划”的时长就是 3 秒钟的话）。下图以“hello”的莫尔斯编码

为例，说明字母之间的停顿（间隙）。

•••• • •—•• •—•• ———

单词之间则通过相当于两个“划”的时长来分隔（如果一个“划”的时长为 3 秒钟，那么这个停顿就应当为 6 秒钟）。例如下图是“hi there”所对应的编码。

•••• •• — •••• • •—• •

手电筒开和关的时长并没有限定。它们都与一个“点”的时长相关，而这依赖于手电筒开关能以多快的速度扳动，以及莫尔斯编码的发送者能以多快的速度记下一个指定字符。对于一个快速发送者，他的“划”的时长，也许只相当于一个慢速发送者的“点”的时长。这会带来一个小问题，即阅读莫尔斯编码信息会变得很困难，但是经过一两个字母之后，接收者通常就可以分清楚哪一个是“点”，哪一个是“划”了。

乍一看，莫尔斯编码的定义与打字机字母的排列一样都是随意的。这里的定义是指字母表中的字母与各种“点”和“划”的组合序列之间的对应关系。然而，仔细研究之后，我们就会发现事实并非完全如此。简单且短促的编码，被分配给字母表中使用频率较高的字母，例如 E 和 T。拼字比赛选手和“幸运轮”的粉丝们可能会很快意识到这一点。而不太常用的字母，比如 Q 和 Z（这只会让选手在拼字中得到 10 分）则被分配以较长的编码。

几乎每个人都知道一点儿莫尔斯编码。三个点、三个划，再加三个点就表示 SOS，即国际求救信号。SOS 不是一个缩写，这只是一个易于记忆的莫尔斯编码序列。在第二次世界大战期间，英国广播公司一些无线电广播以贝多芬第五交响曲的片段来作为节目的前奏——BAH，BAH，BAH，BAHMMMMM，而贝多芬绝不会想到他所谱写的这段乐曲可以表示为莫尔斯编码的 V，即胜利（Victory）之意。

莫尔斯编码的一个缺点是，它没有区分大写字母和小写字母。但是除了可以代表字母外，莫尔斯编码还使用一组由点和划组成的五元序列来表示数字，如下所示。

1	· — — — —	6	— · · · ·
2	· · — — —	7	— — · · ·
3	· · · — —	8	— — — · ·
4	· · · · —	9	— — — — ·
5	· · · · ·	0	— — — — —

这些编码至少比字母编码更有规律一些。大部分标点符号由5个、6个或者7个“点”和“划”的组合序列来表示，如下所示。

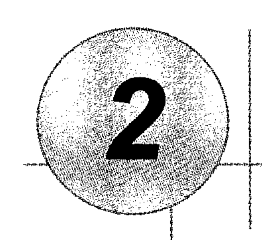
.	· — · — · —	'	· — — — — ·
,	— — · · — —	(	— · — — ·
?	· · — — · ·	)	— · — — · —
:	— — — · · ·	=	— · · · —
;	— · — · — ·	+	· — · — ·
-	— · · · · —	$	· · · — · · —
/	— · · — ·	¶	· — · — · ·
"	· — · · — ·	_	· · — — · —

还有一些编码则用来表示某些欧洲语言中的重音字母，以及用于特殊目的的速记序列。如SOS的编码就是这样的一个速记序列：必须连续地传递这三个字母，而且字母之间只有一个点的停顿时间。

如果有一个专门为此设计的手电筒的话，你会发现和朋友之间通过莫尔斯编码交流将变得更加简单。除了一个可以滑动的开关外，这种手电筒还有一个按钮开关用来控制灯光的明灭。在一些实际应用中，可以达到每分钟传递5到6个单词的速度——虽然这仍比讲话慢得多（讲话的速度在每分钟100个单词左右），但这已经足够了。

当你和朋友最终熟记了莫尔斯编码之后（这是能熟练地收发编码的唯一方法），你们甚至可以在口语中使用它，用来取代正常的语言。为了使编码发送的速度最快，你可以把“点”读作“嘀（dib）”，把“划”读作“嗒（dab）”。文字也可以用同样的方式简化成“点”和“划”的序列，莫尔斯编码的口语版把讲话内容缩减到只剩下两个声音了。

问题的关键就在于数字2。两种闪烁，两种声音。事实上，两个不同的事物，只要经过适当的组合，就可以表示所有类型的信息，这的确是千真万确的。



编码与组合

莫尔斯码（Morse Code）是由塞缪尔·莫尔斯发明的（1791-1872），在本书的其他章节中我们还将频繁地提到他。莫尔斯码其实是伴随着电报机的问世而被发明的，关于电报机，我们在后面也将做详细的探讨。正如通过研究莫尔斯码我们可以很方便地理解编码的本质一样，通过电报机来了解计算机硬件也是个不错的途径。

大多数人都会发现莫尔斯码的发送比接收更为简单。即使你并没有熟记莫尔斯码，也可以很方便地使用下面这张按字母表顺序排列的表格。

A	•—	J	•———	S	•••
B	—•••	K	—•—	T	—
C	—•—•	L	•—••	U	••—
D	—••	M	——	V	•••—
E	•	N	—•	W	•——
F	••—•	O	———	X	—••—
G	——•	P	•——•	Y	—•——
H	••••	Q	——•—	Z	——••
I	••	R	•—•		

比起发送莫尔斯码，接收编码并进行解码要费时费力得多，因为译码者不得不根据

一串由“点”、“划”组成的晦涩的编码序列来反查字母。例如，如果你接收到一串形如“划-点-划-划”的编码，那么你就必须从表的第一个字母开始逐个搜寻，直到找到与这串编码相符的字母“Y”为止。

问题就出在这里，因为我们现在只有一张提供“字母 → 莫尔斯码”的编码表，而缺少一张可以实现反向查询的“莫尔斯码→字母”译码表。在开始学习莫尔斯码的初级阶段，如果有这样的一个表无疑将是很方便的。但是要建立这样一张表，谈何容易。似乎这些字母对应的“点-划”序列并没有什么规律。

所以忘掉字母序列吧。或许根据编码中所包含点、划的多少来对其进行分组，是一个更好的组织这些编码的方法。例如，一个仅包含一个点或一个划的莫尔斯码只能代表两个字母：“E”或“T”。

一组含有 2 个点或划的编码组合，可以给我们呈现出 4 个字母——I，A，N 和 M。

·	E
—	T

··	I	—·	N
·—	A	——	M

一组含有 3 个点或划的莫尔斯码可以为我们表示更多的字母。

···	S	—··	D
··—	U	—·—	K
·—·	R	——·	G
·——	W	———	O

最后（如果我们不想考虑存在数字和标点符号的莫尔斯码的情况），一串由 4 个点或划组成的莫尔斯码就可以表示 16 个字符。

····	H	—···	B
···—	V	—··—	X
··—·	F	—·—·	C
··——	Ü	—·——	Y
·—··	L	——··	Z
·—·—	Ä	——·—	Q
·——·	P	———·	Ö
·———	J	————	Ş

综合以上数据来看，这四张表包含了 2+4+8+16 组码字，总共表示了 30 个字母，比拉丁字母表的 26 个字母还要多出 4 个。所以，你会注意到最后一个表中有 4 组编码是用来表示重音字母的。

当有人给你发送莫尔斯码的时候，上述四张表可能会让你的解码工作变得轻松很多。当接收到某一代表特定字母的码字后，你就可以知道其中所包含的“点”和“划”的数目，那么你至少可以很快找到对应的表格去进行查找。每个表格都组织得很规整，全部是“点”的码字被排在左上角，而全部是“划”的则被排在右下角。

你发现这四张表格在大小上的规律了么？注意看，每个表格所包含的码字数目都是前一张的两倍。这其实很好理解：每个表格所含有的码字，可以看成是在前一张表格所包含的全部码字上再加一个“点”，或者再加一个“划”而组成的新码字。

我们可以用如下这样一个列表来总结这个有趣的规律。

点和划的数目	码字的数目
1	2
2	4
3	8
4	16

在这四个表中，每张表的码字数都是前一张表码字数量的两倍，因此如果第一张表含有 2 个码字，那么第二张表则含有 2×2 个码字，而第三个表就有 2×2×2 个码字。下面用另一种方式呈现这个规律。

点和划的数目	码字的数目
1	2
2	2×2
3	2×2×2
4	2×2×2×2

当然，如果我们遇到了数字的自乘，就可以通过幂的方式来表示它。例如，2×2×2×2 可以记作 2^4（2 的 4 次幂）。数字 2、4、8 和 16 都是以 2 为底数的幂值，因为你可以通过使其自身乘 2 来得到它们。由此我们的总结列表也可以写成下面这种样子。

点和划的数目	码字的数目
1	2^1
2	2^2
3	2^3
4	2^4

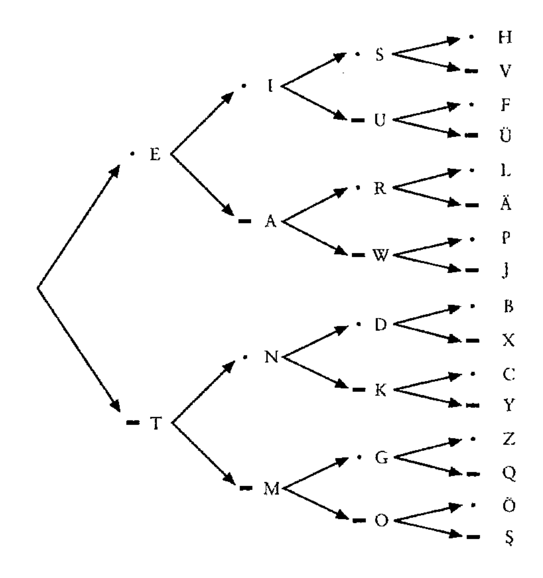
现在这个表已经变得很简洁了。如果知道了码字中“点”和“划”的数目，那么以这个数目为指数的 2 的幂运算结果就是其总共可以表示的码字数。我们可以用下面这个简单的公式来概括上述表格所表示的内容：

$$\text{码字的数目} = 2^{\text{“点”和“划”的数目}}$$

使用 2 的幂值的形式可以表示很多码字，在下一章中，我们还将接触另外一个例子。

为了让莫尔斯码的解码过程更加简单，或许画张图会有所帮助，例如下面这张树型图。

这张图给出了所有字母及其所对应的由“点”和“划”组成的连续序列。当对一串码字进行解码时，我们需要沿着箭头从左向右进行搜寻。以“点-划-点”的码字为例来说，当你需要找出这串码字所代表的字母时，应首先从图的左边开始，选择“点”的分支；然后继续沿着箭头向右选择“划”，接着又是一个“点”。找到最后一个“点”时结果就会紧随其后出现了，没错就是字母“R”。



如果仔细想一想，你就会发现构建这样一个表对于定义莫尔斯码规范来说是很必要

的。首先，它确保了我们不会对不同的字母定义相同的码字。其次，通过这个表我们可以用尽可能短的码字来表示所有的字母，而避免产生编码长度上的浪费。

我们可以继续加长码字至 5 位或者更长，不过这可能超出页面打印边界。一串由 5 个“点”或“划”组成的编码串可以为我们提供 32（$2\times2\times2\times2\times2$，或 2^5）种扩展的码字。对于莫尔斯码中定义的 10 个数字和 16 个标点符号来说，通常这已经足够了，而实际上数字确实就是使用 5 位的莫尔斯码来表示的。但是在很多其他编码方式中，5 位码字常用来表示重音字母而不是标点符号。

为了把所有的标点符号也都包含进去，编码系统必须要扩展到 6 位了！扩展后将为我们提供 64（$2\times2\times2\times2\times2\times2$，即 2^6）种新增的码字，这样总共的码字就达到了 $2+4+8+16+32+64$，也就是 126 种！这对莫尔斯码来说有点太多了，甚至还留下了很多“未定义”的码字。这里“未定义”用来表示那些不代表任何字符的码字。如果你在接收莫尔斯码的时候收到了一个未定义码字，可以肯定发送方一定是出了差错。

我们很容易就能得到这样一个小公式：

$$\text{码字的数目} = 2^{\text{编码的位数}}$$

利用它就可以继续计算出更长位数的点划序列所能表示的码字数目了。

点划的数目	码字的数目
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
4	$2^4=16$
5	$2^5=32$
6	$2^6=64$
7	$2^7=128$
8	$2^8=256$
9	$2^9=512$
10	$2^{10}=1024$

幸运的是，我们并不需要写出所有可能的码字来计算码字的总数目。我们需要做的只是让 2 不断地与自己相乘。

莫尔斯码也被称作二进制码（Binary Code），因为这种编码的组成元素只有两个——“点”和“划”。这跟硬币有些类似，因为硬币落到地上只能是正面朝上或反面朝上。二元对象（例如硬币）和二进制码（例如莫尔斯码）常常使用 2 的乘方来进行描述。

上面所做的关于二进制编码的分析工作，其实是数学的一个分支，称作“组合学”或“组合分析”，而我们所作的分析则只能说是一个简单的练习。传统意义上来说，因为组合分析涉及类似像扔硬币、掷骰子这样的需要对其组合数目进行推算的问题，所以它经常被应用到概率和统计学中。但是它对于我们理解码字的组合与分解也是十分有帮助的。

3

布莱叶盲文与二进制码

塞缪尔·莫尔斯并不是第一个将书面语言的字母成功编码的人。实际上，在因自己的名字被用来命名某种编码方式而名声大噪的人中，莫尔斯先生也不是第一人。这项荣誉应该归一个法国盲人小伙子所有。尽管他比莫尔斯要晚出生18年，但是他很早就创建了自己的编码规范。他的生平鲜为人知，尽管关于他的故事流传至今的为数不多，但这也足以构成一个引人注目的传奇。



路易斯·布莱叶（Louis Braille）于1809年生于法国的库普雷，这是一个距离巴黎市东只有25英里的小镇。布莱叶的父亲是一个马具匠人。在布莱叶3岁的时候——一个本不该在父亲的工作间玩耍的年纪——他不小心被一个尖锐的工具刺伤了一只眼睛。伤口感染也影响到了他的另外一只眼睛，最后使他的双目完全失明。按常理，布莱叶将在无知和贫困当中度过自己的一生（就像当时大部分盲人那样），但是小路易斯对知识的渴望和过人的智慧很快被

人们所发现。他最初同其他孩子一样在镇上的小学上学，后来在他 10 岁的时候，在小镇牧师和一名学校老师的帮助下，布莱叶被送往巴黎皇家盲人学校学习。

很显然，盲人教育中一个主要的障碍就是盲人无法阅读印刷的书籍。瓦伦丁·霍伊（1745-1822），巴黎皇家盲人学校的创始人，曾发明过一种在纸面上印下凸起文字的系统，这样盲人就可以通过触摸的方式来进行阅读了。但是这种文字系统使用起来很困难，而且使用这种方法来印刷的书籍也非常少。

霍伊先生视力健全，因此他被自身的感知模式所禁锢。对他来说，一个字母 A 就是一个 A，而且字母 A 在记录时也必须看起来（或者感觉起来）像一个 A（如果让他用手电筒交流的话，估计他会像我们最开始所做的那样，试图在空中比划出要表达的字母，但其实我们发现这法子根本不灵）。霍伊先生或许没有意识到，有一种完全不同于印刷字母的文字系统可能更加适合盲人阅读。

这种非常规编码方式的起源说出来可能有些出人意料。查尔斯·巴比尔（Charles Barbier），法国军队的一位军官，在 1819 年发明了一种他自称为“écriture nocturne”（也叫“夜间书写”）的文字系统。他在厚纸上使用凸起的点和划的组合来表示文字，这样当部队需要无声交流的时候，即使光线很暗，士兵们也可以通过这些符号互相传递信息。他们使用一个锥形的铁笔在厚纸的背面书写，这样纸的正面就会有相应的凸起。然后人们就可以使用手指触摸这些凸起的点和划来进行阅读了。

巴比尔文字系统的缺点是太过复杂了。该系统并非使用与字母表相对应的点划编码串来表示字母，而是用与读音相对应的编码串表示，因此有时仅仅是为了表示一个单词，就不得不使用很多的码字。如果只是传递简短的消息，这个系统用起来倒还不错；但是在表示长文本的时候，就明显力不从心，就更别说用来对整本书进行编码了。

布莱叶在 12 岁的时候就很熟悉巴比尔的这种文字系统了。他特别喜欢使用凸起的点，不仅仅因为凸起的点通过手指就可以很容易实现阅读，还因为它们“写”起来也很简单。在教室里，一个盲人学生如果有了纸张和铁笔，他就可以做笔记，而且同时还能阅读记下来的文字。路易斯·布莱叶开始不辞辛劳地改进这个文字系统，3 年以后（那时他 15 岁）就创建成了自己的系统，而这个系统中的一些基本规范，直到今天仍在被人们所使用。在很长一段时间内，这种新的文字系统只被他们学校内部的人们所熟悉，但是渐渐地，布莱叶盲文传播到了世界的各个角落。1835 年，路易斯·布莱叶患上了肺结核。1852

年，就在布莱叶 43 岁生日刚过完不久，病魔无情地夺走了他的生命。

如今，在引导盲人进入文字殿堂的道路上，改进后的布莱叶盲文系统与磁带录音书进行着竞争，但是布莱叶盲文仍然是不可替代的工具，特别是对于又聋又盲的人来说，布莱叶盲文仍然是他们开启阅读大门的唯一钥匙。近些年来，布莱叶盲文被越来越多地应用到公共场所中，使得盲人朋友们可以越来越方便地使用电梯、自动取款机等设备。

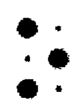
在这一章中我们将解析布莱叶盲文，来看看它是如何工作的。我们并不是要真的学习布莱叶盲文，而且也无须刻意记住什么关于它的内容。我们仅仅希望从中窥探到编码的一些本质。

在布莱叶盲文中，每个在书写文字中用到的符号——具体来说就是字母、数字和标点符号——都被编码成为 2×3 的点码单元中的一个或者多个凸起的点。这个点码单元包含的点通常使用 1 到 6 的数字来编号。

1 ○ ○ 4
2 ○ ○ 5
3 ○ ○ 6

在现代应用中，使用特殊的打印机或轧花机可以将布莱叶盲文印到纸张上。

即使只是用布莱叶盲文在本书中印上几页，造价也实在是有点儿高，所以我在书中使用一种符号来表示布莱叶盲文。在这种表示方法中，点码单元中所有的 6 个点都会被表示出来。大点表示这是一个凸起的点，小点则表示其对应的位置是平的。例如，下面的布莱叶盲文：



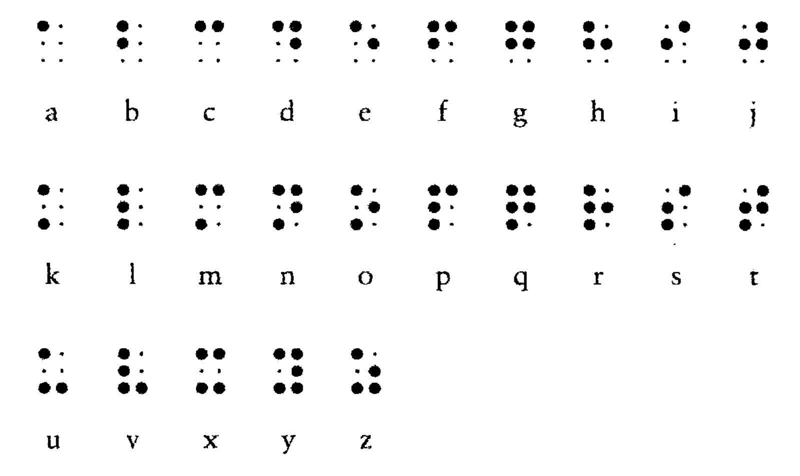
其中，第 1，3，5 点是凸起的，而第 2，4，6 点的位置是平的。

令我们感兴趣的是，这些点码都是二进制的。一个单独的点不是平的就是凸起的。也就是说我们可以把在莫尔斯编码中学到的知识应用到布莱叶盲文的分析中！我们现在已经知道，每组有 6 个点，并且每个点有平和凸两种状态，因此 6 个可平可凸的点的组合数就是 $2\times2\times2\times2\times2\times2$，即 2^6，也就是 64。

因此，布莱叶盲文系统能够表示 64 个不同的码字。下图就是所有可能的 64 种码字。

假如发现布莱叶盲文中用到的码字数目少于 64，就会有人问了，为什么 64 个可能的码字中有一些被遗弃不用呢？假如我们发现布莱叶盲文中用到的码字数目超过 64，问题就更严重了，我们会怀疑自己的智商，甚至怀疑基本的数学原理——到底 2 加 2 是不是等于 4。

还是让我们来开始解析布莱叶盲文吧，首先看看基本的小写字母表。



例如，词组“you and me”用布莱叶盲文表示成如下组合：

注意，一个单词中每个字母所对应的点码单元之间都用一小块空白分开；而单词之间则用一个大的空格（实际上是没有凸点的点码单元）所分隔开来。

这就是路易斯·布莱叶制订的布莱叶盲文的基本规范，至少它对于拉丁字母很适用。布莱叶还制订了表示重音符号字母的编码，这在法语中要经常要用到。注意这里没有“w”这个字母的编码，因为在传统的法语中不会用到它（别担心，这个字母最后会出现的，下文将作讲解）。到此为止，64 个码字中只使用了 25 个。

经过仔细的检查，你会发现，从我们列举的那个三排布莱叶盲文的例子（小写字母表）中，可以总结出一个规律。第一排（字母 a 到 j）只用到了点码单元中最上面的四个点——第 1、2、4 和 5 点。第二排在复用了第一排的编码的基础上，把第 3 点改为凸点。第三排也沿用了同样的规律，只是将第 3 和 6 点改为凸点。

自从路易斯·布莱叶发明布莱叶盲文以来，其应用已经扩展到各个领域。目前在英文出版物中最常用的盲文系统被称为二级布莱叶盲文（Grade 2 Braille）。二级布莱叶盲文使用了很多缩写，以便于保存树型结构和提高阅读速度。例如，如果字母的码字单独出现，它们就表示一个普通的单词。以下三排图样（包含“完整的”第三排）为我们展示了这些单词的码字。

(none) but can do every from go have (none) just

knowledge like more not (none) people quite rather so that

us very it you as and for of the with

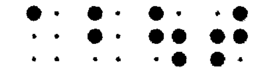
因此，短语“you and me”使用二级布莱叶盲文就可以表示为：

到目前为止，我们已经描述了 31 个码字——单词间的大空格（即没有凸点的点码单元）以及总共 3 排每排 10 个的字母和单词码字。与理论上可以达到的最多码字数——64 相比，我们还差得远呢。正如从下面的分析中我们将看到的那样，在二级布莱叶盲文里，没有任何的码字会被浪费。

首先，我们可以使用字母 a 到 j 的码字加 6 号凸点的组合。这些新码字通常用来表示单词中字母串的缩写，还有我们前面所担心的 w 字母（像上文的二级布莱叶盲文单词编码一样，表示 w 字母的编码也可以表示一个单词）。

ch　gh　sh　th　wh　ed　er　ou　ow　w
（will）

例如，单词“about”使用二级布莱叶盲文可以记作：



然后，我们取从 a 到 j 的码字，“降低”它们使用的点位，只用到编号为 2、3、5 和 6 的点，这样就得到了新的码字。根据上下文环境，它们将被用来表示一些标点符号或者字符串缩写。

ea	bb	cc	dis	en	to	gg	his	in	was
,	;	:	.		!	()	“		”

前 4 个码字分别表示逗号、分号、冒号和句号。需要注意的是，左右括号使用的是相同的码字，但是开闭引号使用的却是不同的码字。

目前为止我们已经定义了 51 个码字了。“占用 3、4、5 和 6 号点”的码字还有很多组没有被定义，下面的 6 组就是这些被遗漏的码字，我们也用它们来表示一些字符串缩写和其他的标点符号。

st　ing　ble　ar　'　com

/　　#　　-

表示字符串缩写“ble”的码字很重要，因为当它不是用来作为单词的一部分时，它的出现就意味着之后的码字应被译为数字。表示数字的码字与表示字母 a 到 j 的码字是相同的。

1　2　3　4　5　6　7　8　9　0

因此，下面这组编码：

表示的就是数字 256。

如果你还要继续下去，那么我们还需要定义 7 个码字才算完全用满原有的 64 个码字。它们是：

第一个码字（4 号点凸起）用来表示音调符号（accent indicator）。其他的码字则用来表示缩写的前缀或者其他的功能：4 号和 6 号点凸起的码字（上表的第 5 个码字），根据上下文用来表示小数点或者强调符号（emphasis indicator）。5 号和 6 号点凸起的编码，表示与“数字标识符”相对应的“字母标识符”（letter indicator）。

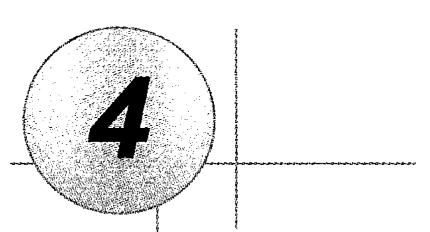
最后（如果你现在还为布莱叶盲文如何表示大写字母忧心忡忡的话），我们还有 6 号凸点——大写标识符。这个符号表示它之后的字母将都被译为大写。例如，我们可以把这套文字系统的原创者的名字记做：

这串编码分别表示了一个大写字母标识符、字母 l、缩写字符串 ou、字母 i 和 s，以及一个空格，再加一个大写字母标识符，还有字母 b、r、a、i、l、l 和 e。（在实际应用中，

可能会把名字末尾不发音的 2 个字母省略掉，以进一步压缩编码）

综上所述，我们将 6 位二进制码（其实是 6 个点）所能表示的全部 64 种可能的编码都罗列了一遍。而且这 64 组编码中有很大一部分，根据上下文的不同将有着双重身份。尤其值得注意的是数字标识符和取消“数字标识状态”的字母标识符。它们改变了后面编码的意义——从表示字母到表示数字，又从表示数字回到表示字母。像这样的编码通常被称作“优先码”（precedence codes）或者“换档码”（shift codes）。它们改变着作用域内编码的含义，直到作用域结束。

大写字母标识符表示紧随它的字母（而且仅仅是紧随它的字母）应该被译为大写。类似这样的编码被称为“逃逸码”（escape codes）。逃逸码让你“逃离”对编码串单调的、一成不变的解析，而转入一种新的解析方式中。在以后的章节中我们将看到，在使用二进制码对书面语言进行编码时，换档码和逃逸码是相当常见的。



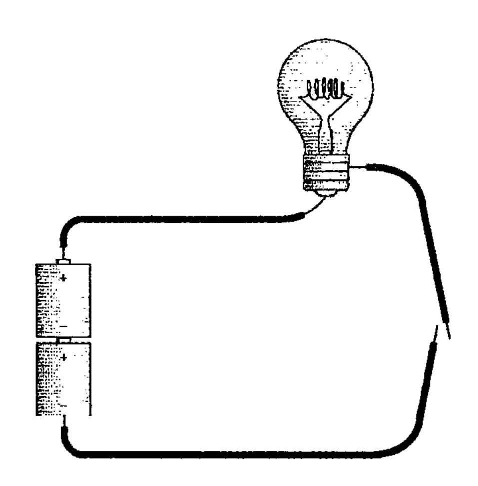
手电筒的剖析

手电筒的应用是极其广泛的，例如它可以帮助人们在背光的环境下阅读，也可以用于发送编码后的信息，这只是其中最显而易见的两个功能。普通的家用手电筒，还可以作为科普教育中的重要道具，引导人们走进神秘的电的世界。

电是一种神奇的现象。尽管电已经被普遍地应用到各个领域中，但当人类自豪地宣称已经理解了电的工作机制时，真实情况是，关于电仍存在着大量的未解之谜。但是，在本书中恐怕我们无论如何都得仔细钻研一下电学了，幸好，我们只需要一点基本概念就可以理解电在计算机中的工作原理。

手电筒无疑是大多数家庭里最简单的家用电器。拆开一个标准的手电筒，你会发现里面有一对电池、一个灯泡、一个开关、一些金属片，还有一个可以容纳这些元件的塑料外壳。

你只需要电池和灯泡就可以自己来做一个简单的手电筒了。当然，你还需要一些短的绝缘导线（末端剥去绝缘皮），此外你还必须把这些元件连接起来，如下图所示。



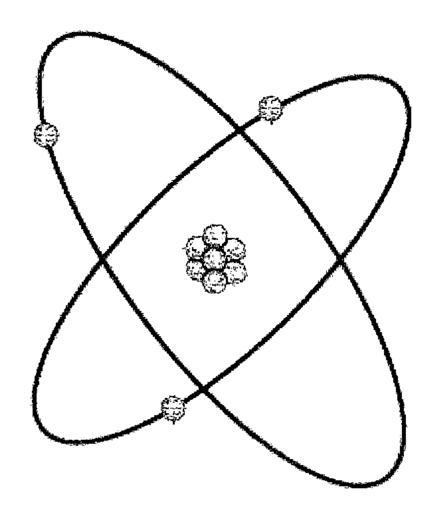
注意，在上图的右边，我们可以看到两个断开的线头，那就是开关。假设这两个电池没有问题，灯泡也没有烧坏，让这两个线头互相接触，就可以点亮灯泡。

刚刚完成的就是一个简单的电路。首先要注意的是，一个**电路**就是一个**环路**。只有从电池到导线，再到灯泡和开关，然后再通过导线回到电池的整个回路是连通的，灯泡才能被点亮。电路中出现任何的中断都将使灯泡无法正常发光。开关的作用就是控制这个断开的过程。

电路这种环状回路的特性说明电路中有某种东西在循环流动，或许有些像水流过水管那样。“水和水管”这个比喻经常被用来解释电流工作的原理，但是这种模型就如同所有其他的类比一样最终也无法自圆其说。在茫茫宇宙中，电不同于任何其他的事物，我们必须用它的术语来解释它。

研究电流工作原理的、最主流的科学理论叫做“电子理论”（electron theory），这套理论认为电流是由电子的运动而产生的。

众所周知，所有的物质——我们所能看到和感觉到的事物——（通常情况下）都是由叫做原子（atom）的极其微小的东西组成的。每一个原子又由三种粒子构成：它们分别是中子（neutron）、质子（proton）和电子（electron）。你可以把原子的结构画成一个小太阳系，其中中子和质子被束缚在原子核内，而电子则围绕着原子核旋转，犹如行星围绕太阳旋转一样。



我要提醒大家的是，如果你能搞到一个放大倍数很大的显微镜足以看清原子结构的时候，你会发现这张图与原子实际的样子并不一样。但是我们可以把它当做一个研究模型，方便我们进行一些探讨。

上图所表示的原子包含有 3 个电子、3 个质子和 4 个中子，这表明它是一个锂原子。锂是 112 种已知元素之一，这些元素都有一个 1 到 112 之间的特定原子序数（atomic number）。原子序数表明了这种元素一个原子的原子核中所含的质子数，（通常）同时也是一个原子所含的电子数。锂的原子序数是 3。

原子之间可以通过化学的方式结合形成分子（molecules）。分子的性质通常与组成它的原子大相径庭。例如，水是由水分子组成的，每个水分子由两个氢原子和一个氧原子（即 H_2O）构成。很明显，水跟氢气或氧气都是截然不同的。同样，食盐的分子是由钠原子和氯原子构成的，显然这两种东西都没法让我们的炸薯条变得更加可口。

氢、氧、钠和氯都是元素。水和盐都是化合物（compound）。不过盐水是混合物（mixture）而不是化合物，因为水和盐都各自保留着它们自己的性质。

一个原子中电子的数目一般情况下与质子数目相同。但是在某些情况下，电子可能从原子中脱离。这就是电流产生的原因。

电子（electron）和电（electricity）这两个单词都是起源于古希腊文“ηλεκτρον（elektron）”，关于这个词，你可能会猜测它是表示“微小而又无形的东西”之类的意思。

但是事实并非如此——ηλεκτρον 的意思是“琥珀（amber）”，它是树的汁液硬化后变成的一种玻璃状固体。这两种风马牛不相及的东西会被联系到一起的原因源于古希腊人所进行的试验，他们曾经通过琥珀摩擦羊毛，产生了我们称之为“静电”的东西。用琥珀在羊毛上摩擦使得羊毛掠夺了琥珀的部分电子。结果羊毛会因为电子数超过了质子数而卷曲，而琥珀的电子数比质子少了。后来在现代的其他一些实验中，人们发现，毛毯也可以从我们的鞋底中带走电子。

质子和电子都具有带电荷（charge）的性质。质子有一个正电荷（+），电子有一个负电荷（-）。中子是中性的，因而不带电荷。尽管我们使用了加号和减号来标识质子和电子，但是其实这些符号并不带有任何的数学方面的含义，也就是不能说明质子拥有电子所没有的性质。使用这些符号仅仅表示质子和电子在某个方面上的性质是相反的。这个相反的特性也正表明了质子和电子是相互关联的。

当质子和电子在相同数目的条件下共存时，它们都处于最和谐、最稳定的状态。如果质子和电子之间出现失衡现象，它们就会试图进行自我修复。当地毯偷偷摸摸地从你的鞋子上挖走电子后，一切会在你触摸到一些东西而感到被电了一下时，又回归到平衡状态。静电火花是电子运动引起的，是电子通过一个回路——从地毯传到你的身体，再回到鞋子中的过程所造成的。

质子和电子之间的关系还可以这样描述，就是异性电荷相吸引，同性电荷相斥。不过关于这一点，我们仅仅靠观察原子结构图是观察不出来的。原子核中的质子被一种力量束缚到一起，这种引力要强过同性电荷之间的斥力，我们称之为“强力”（strong force）。强力有可能会引起原子核的分裂，而核能就是由此产生的。在本章中，我们仅仅讨论通过电子得失来获得电能的问题。

静电不仅仅存在于手触摸门把手时产生的小小火花中。在风暴中，底层的云积聚了大量电子而顶端的云失去电子；最后，一道闪电划破长空，使这一切又回到平衡。闪电是大量电子从一端快速地移动到另一端所形成的。

手电筒电路中的电流显然要比火花或者闪电更容易驾驭得多。灯泡之所以能稳定而持续地发光，是因为电子并不是简单地只是从一点跳到另一点。电路中，某原子所含有的一个电子逃逸到它相邻的下一个原子中，与此同时，这个原子又从相邻的上一个原子中获取一个电子，而失去电子的原子又会从与其相邻的一个原子获得电子，如此循环。

电路中的电子不断地从一个原子移动到下一个原子，就形成了电流。

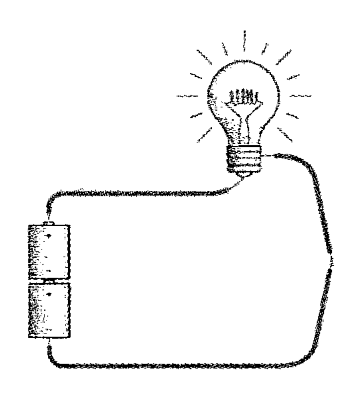
这一切都不是自发产生的。我们不能仅仅只把一堆破旧的电路材料随便连接起来后就开始祈求电流的出现。我们需要一种设备来促成电路中电子的流动。回头再分析一下前面所画的简单手电筒的线路图，我想我们能很有把握地断定，使电子流动的既不是导线，也不是灯泡，那么或许这个关键的设备就是电池。

关于手电筒中用到的电池类型，大部分人应该都有所了解。

- 它们是筒状的，并且有不同的大小，例如 D、C、A、AA 和 AAA 等型号。
- 无论电池的大小如何，都标有“1.5 伏”。
- 电池的一端是平的，标有一个负号（–）；另一端有一个小的凸起，并标有一个正号（+）。
- 如果你想电器正常工作的话，装电池的时候，就要注意让它的“+”端朝着正确的方向。
- 电池的电能终将被用尽。有些电池可以进行再次充电，但是有些却不可以。
- 最后，我们猜测，通过某种很神奇的方式，电池能够自己产生电能。

所有电池的内部都会发生化学反应，也就是说一些分子被分裂形成其他的分子，或者分子间互相结合形成了新分子。电池内的化学物质是经过研究精心选择的，它们之间的化学反应能够使多余的自由电子聚集到标负号“–”的那端（称为负极或者阴极），而在标有正号“+”的那端（称为正极或者阳极）则变得急需额外的电子。于是，化学能就被转化成了电能。

这种化学反应不会一直进行下去，除非我们有办法能使负极多余的自由电子回到正极去。因此如果我们没有给电池连上任何东西，什么事情也不会发生（实际上电池内仍有化学反应发生，但是非常缓慢）。电路可以将负极的电子带走，以补充正极缺失的电子，只有当存在电路的时候，化学反应才会发生。如下图所示，电子在电路中的这次旅行是沿逆时针方向的。

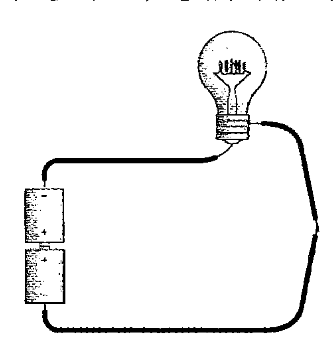


在本书中，红色[1]用来表示在导线中有电流经过。

如果不是基于这样一个简单的事实——所有的电子，不管它们在哪里，都是完全一样的，或许电池内化学物质中的电子就无法如此自由地与铜导线中的电子混合在一起。铜的电子与其他物质的电子没有任何差别。

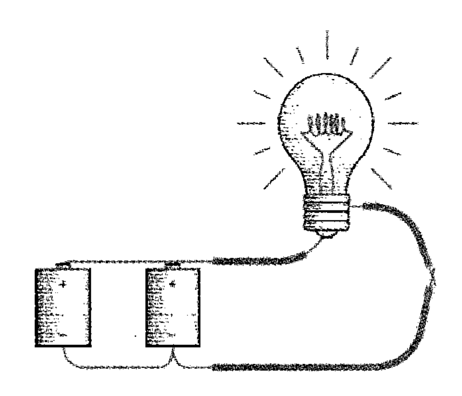
注意，所有的电池都是按相同方向放置的。下面的电池的正极，从上面电池的负极获得电子。就好像两个电池组合成了一块更大的电池，它的正负极分别在两块小电池的两端上。组合后的电池不再是 1.5 伏，而是 3 伏。

如果我们把电池组中的一块电池倒置，电路就不能工作了。



两个电池的正极都要为化学反应提供电子，但是由于这两个正极相互接触，电子根本无法通过某种途径到达它们。如果两个电池的正极连到了一起的话，它们的负极也应该连在一起，如下图所示。

1 由于黑白印刷的原因，在书中红色只能显示为淡灰色。——译者注



现在电路能够正常工作了。与前文我们提到的“串联”方式不同，这种连接电池的方式被称做“并联”。并联后的电压是 1.5 伏，与每个电池的电压是一样的。或许灯泡仍能发光，但是肯定没有串联电池的灯泡亮。不过电池的使用寿命也将延长一倍。

一般情况下，我们认为电池为电路提供了电能。但是我们也同样可以反过来想，电路为电池内的化学反应的进行提供了条件。电路把电子从电池的负极运走，然后转移到电池的正极。电池中的化学反应将持续进行，直到所有的化学物质被消耗完，那时你就可以把电池扔掉或者重新给它充电了。

从电池的负极到正极，电子流经了导线和灯泡。但是为什么我们需要导线呢？电流不可以直接被空气传导吗？哦，可以说能，也可以说不能。电流可以通过空气传导（特别是潮湿的空气），不然的话我们就看不到闪电了。但是电流也不能轻易地穿过空气。

一些物质在导电能力上要比其他一些物质明显更好一些。元素的导电能力与它原子内的结构有关。电子围绕原子核运行的轨道分为不同等级，我们称其为“电子层”。如果原子在最外电子层中只含有 1 个电子，那么这个电子很容易逃逸，这就是易导电物质所需具备的特性。这些物质对于电流来说是“导通”的，因而被称做导体（conductor）。最好的导体是铜、银和金。这三种元素位于元素周期表的同一列不是巧合。其中铜是用来制作导线的最常见的原料。

与导电性相反的是阻抗性。有一些物质与其他的物质相比更不容易让电流通过，我们称其为电阻。如果一种物质有着很强的阻抗性——也就是说它几乎不能传导任何电流——它就被称为绝缘体（insulator）。橡胶和塑料都是很好的绝缘体，因而它们经常被用来包裹金属导线。布料和木头在干燥的空气中也是很好的绝缘体。不过事实上只要有足够高

的电压，任何的物质都是可以导电的。

铜具有很低的阻抗，但实际上或多或少仍然会存在一点。导线越长，它的阻抗就越高。如果你用几英里长的导线来为自制手电筒布线的话，导线所产生的阻抗将变得很高，以至于手电筒将无法正常工作。

导线越粗，它的阻抗就越低。这可能有些违背我们的直观感觉。你可能会想，粗一些的导线需要更多的电流来“填满它”。但是实际上粗一些的导线可以使更多的电子顺畅地通过线路。

前面我多次提到了电压，但是却还没有定义它。一个电池有 1.5 伏的电压，这意味着什么呢？实际上，电压——因亚历山大·伏特（Alessandro Volta，1745-1827）伯爵的名字而得名，他于 1800 年发明出第一枚电池——在初等电学中是最难理解的概念之一。电压表征了电流做功的“势”（potential），也就是电势能的大小。不管电池是否被连接到电路中，电压都是存在的。

想象一块砖，当它在地板上时，这块砖只有很少的势能。把它从地面上举到离地板四英尺的高度，现在这块砖就会有比较多的势能。如果你想了解所谓“势能”到底是多少，把砖扔掉就可以了。当我们拿着砖跑到一座高楼的楼顶时，它的势能会更多。在这三个场景中，你只是拿着这块砖，它并没有做什么，但是这块砖的势能却差别迥异。

电学中还有一个比较简单的概念是电流（current）。电流与流经电路的电子数有关。它的计量单位是安培，这得名于安德烈·玛丽·安培（André Marie Ampère，1775-1836），不过大家一般简称这个单位为“安”，例如“10 安的保险丝”。假如要获得 1 安的电流，你就需要保证每秒有 6,240,000,000,000,000,000 个电子通过电路中的某一点。

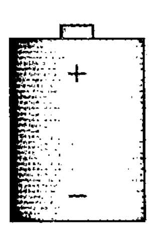
“水和水管”的这个比喻现在可以帮你理解这些概念：电流可以看成是水管中流水的量。电压可以看做是水压。阻抗有些类似水管的宽度——水管越细，阻抗越大。所以水压越大，流经管子的水也就越多。水管越细，水管里流动的水也就越少。水管中流水的量（相当于电流）与水压（相当于电压）成正比，与水管的纤细程度（相当于电阻）成反比。

在电学中，如果你知道了电压和电阻，就可以计算出电路中的电流是多少。电阻——一般来说物质都倾向于阻拦电子的通过——的单位是欧姆，它得名于乔治·西蒙·欧姆（George Simon Ohm，1789-1854）。著名的欧姆定律就是由他提出的。该定律可以表示为：

$$I=E/R$$

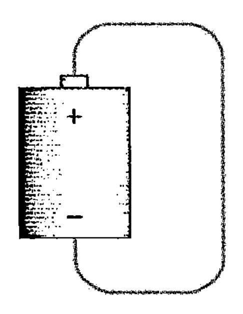
其中，I 用来表示电路中的电流，E 用来表示电压（它代表电动势），R 表示电阻。

例如，下面这节电池是单独放置的，没有连接任何设备。



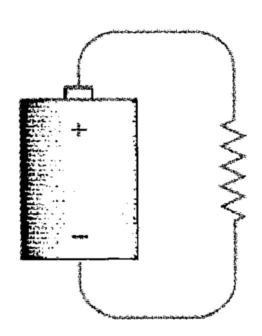
它的电压 E 是 1.5 伏。它代表做功的势能，但是因为正负极通过空气也有连通，所以电阻（R 所代表的物理量）非常非常大，也就是说，电流（I）等于 1.5 伏的电压除以一个极大的数。因而电流在这里几乎是 0。

现在我们用一小段铜线把正负极连接起来（从现在开始，我们不再在示意图上画导线的绝缘层）。



这种情形被称为“短路”。电压仍然是 1.5 伏，但是现在的电阻非常非常小。现在的电流等于 1.5 伏的电压除以一个非常小的电阻值。也就是说电流将变得非常非常大。大量的电子流经导线。在现实情况下，电流会因为电池的型号大小而受到限制。电池或许无法导通如此大的电流，且实际电压也将低于 1.5 伏。如果电池足够大的话，导线将会发热，因为电能被转化成了热能。如果导线继续变得很热，它将会发光甚至熔化。

大部分的电路都介于这两种极端情况之间。我们可以把它们统一表述为如下图所示。



上图中的那段折线对于电气工程师来说是表示电阻的符号。这里我们用它来表示电路中的电阻既不很小也不很大。

如果导线电阻较低的话，它将变热并且发光。这就是白炽灯发光的原理。通常，白炽灯泡公认的发明者是美国著名发明家，托马斯·阿尔瓦·爱迪生（Thomas Alva Edison，1847-1931），但是这种观点是在他取得灯泡的专利之后（1879）被广为传播的，实际上在这个领域很多科学家都有过研究。

灯泡里面有一根很细的金属丝，我们称它为灯丝，一般情况下灯丝是用钨制作的。灯丝的一端连在灯泡底座的凸起上，另一端连到金属底座上，金属底座与凸起之间被绝缘层隔开。金属丝的电阻使它开始发热。如果暴露在空气中，钨丝将达到燃点并开始燃烧，但是在灯泡的真空泡室内，钨丝就会发出光亮。

大多数普通手电筒都用 2 节电池串联成一组，总的电压是 3 伏。手电筒中一般使用电阻为 4 欧的灯泡。因此，电流的大小等于 3 伏除以 4 欧，也就是 0.75 安，也可以写成 750 毫安。这意味着每秒钟有 4,680,000,000,000,000,000 个电子流经灯泡（如果使用欧姆表对手电筒灯泡的电阻进行实际测量，你将得到一个远远小于 4 欧的读数。这是因为钨的阻抗与它的温度有关，当灯泡温度升高并开始发光时，钨丝的阻抗也随之增加）。

你或许已经发现，买回家的灯泡上都标有一个固定的瓦特数。瓦特这个单位得名于詹姆斯·瓦特（James Watt，1736-1819），他以对蒸汽机的研究而广为人知。瓦特是功率（P）的计量单位，它的计算公式如下：

$$P = E \times I$$

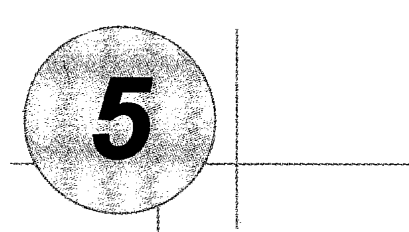
我们的手电筒是 3 伏，0.75 安的，这表明灯泡的功率应该是 2.25 瓦特。

你家里或许在使用 100 瓦的灯泡。它们被设计成能在 120 伏的电压下工作。因此，流经它们的电流的大小等于 100 瓦除以 120 伏，也就是 0.83 安。所以 100 瓦灯泡的电阻是 120 伏除以 0.83 安，即 144 欧姆。

好了，现在我们貌似已经把手电筒里里外外都分析了一遍——电池，导线还有灯泡。但是，我们还忘记了最重要的一部分！

是的，那就是开关。开关控制电路中电流是否接通。当开关被设置成允许电流通过时，我们称它的状态为“开”，或者“闭合”。当处在“关”或者“断开”状态时，开关不允许电流通过。（“闭合”和“断开”这两个词，在用来描述开关时，其含义与用来描述房门时恰好是相反的。一扇闭合的门会阻止所有试图穿过它的东西；一个闭合的开关却可以使电流导通。）

开关只能是闭合状态或断开状态。电流只能是有或者无。灯泡也只能是发光或不发光。就像莫尔斯和布莱叶发明的二进制码一样，这个简单的手电筒要么是开着的，要么是关着的。没有介于二者之间的状态。在后面的章节中，二进制码与电气电路之间的这种相似性将起到很大作用。



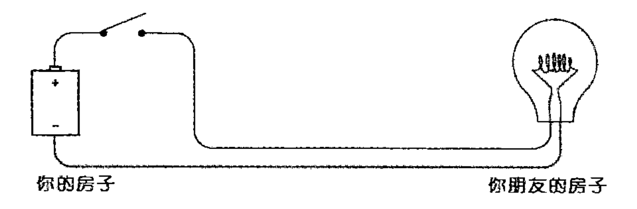
绕过拐角的通信

你 12 岁了。在一个伤感的日子里，你最好的朋友跟随家人搬到别的镇子。从此以后你只能通过电话同他交谈了。但是电话交谈和深夜里用手电筒以莫尔斯码方法会谈是无法相比的。你另外的一个好朋友，也就是你的邻居，现在成了你新的密友。是时候该教给他一些莫尔斯码，让深夜的手电筒再次为你们闪烁了。

可是问题是，这位新朋友的卧室窗户和你卧室的窗户不是对着的。尽管两栋房子挨着，但是卧室窗户是朝向同一个方向的。除非你能想个办法，在室外摆上一面镜子，不然手电筒就不能用来夜谈了。

怎么办?

也许现在你已经懂得一些关于电学的知识了，所以你打算用电池、灯泡、开关和导线来自制一个“手电筒”。在第一次尝试中，你把电池还有开关在卧室中接好，然后从窗户引出两根导线，绕过围墙，接到你朋友的卧室里，在那里把它们接到一个灯泡上。



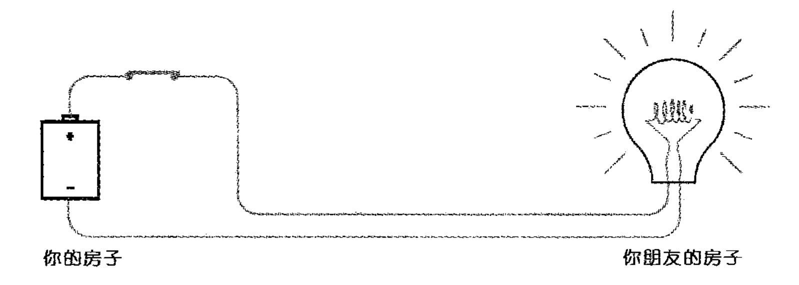
尽管图中只画了一节电池，不过实际应用中你也许要用两个。在以后的示意图中，下面的标志表示一个“关”状态（也称断开状态）的开关。



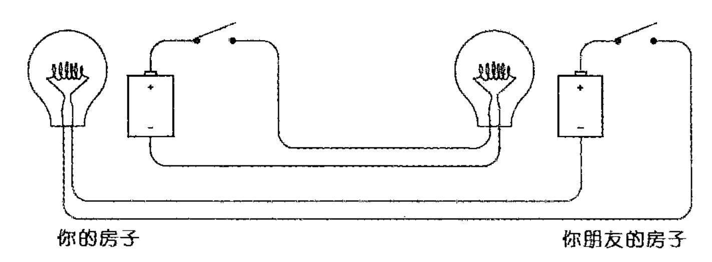
下图则表示开关是“开”的（也称作闭合状态）。



本章手电筒跟上一章中的手电筒的工作方式一样，尽管本章的手电筒中用来连接各个部件的导线比原来长了不少。当你把自己房间那端的开关闭合时，你朋友那儿的灯泡就会被点亮。

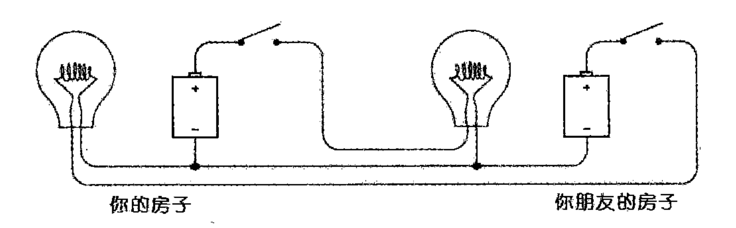


现在，你可以用莫尔斯码来发信息了。既然你已经成功地安装了这个“手电筒”，你就可以照样再安装另一套这样的“远距离”设备，好让你的朋友也能够给你发信息。



恭喜你！你已经架设好了一套双向电报系统！你可能注意到了，这个系统包含了两个相互独立但又完全相同的电路。理论上来说，在你给朋友发送信息的同时，他也可以给你发信息（不过同时阅读和发送电码对你的大脑来说或许是个挑战）。

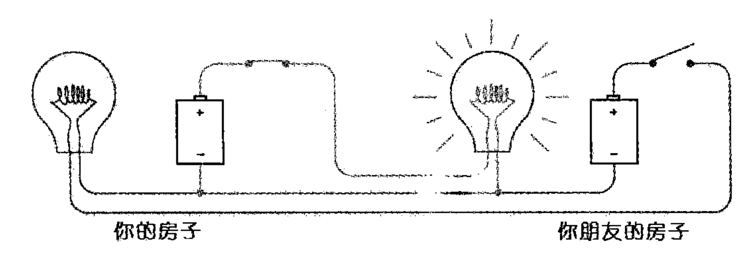
聪明的你也许可以采用如下方式把这套系统加以改进，这样可以节省25%的导线。



注意，现在这两个电池的负极被连在一起。但是这两个回路（电池到开关到灯泡再到电池）依然是相互独立的，尽管现在它们看起来像连体婴儿一样。

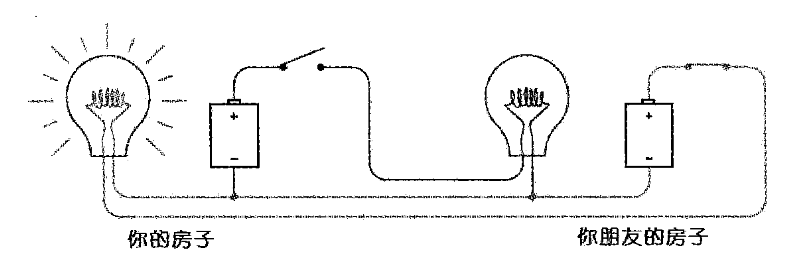
这种接线方式被称为公用连接（common）。这个电路的公用部分，从最左端的灯泡和电池的连接点开始，到最右端的灯泡与电池的连接处结束。我们在图中用两个小圆点把这部分连线标示了出来。

我们来进一步检查一下这个电路，以确保使用时不会有意外发生。首先，当你在自己这端闭合开关的时候，你朋友屋里的灯泡就会点亮。红色的线表示出了电路中电流流动的路径。

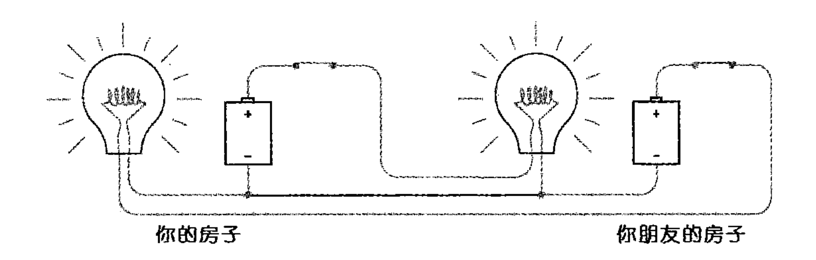


电路的其余部分没有电流通过，因为没有完整路径可以让电子构成回路。

当你没有发送信息，而你的朋友正在发送时，他房间里的开关控制了你房间里灯泡的亮灭。下图的红线仍然表示电流流经的路径。



当你和你的朋友同时发送信息时，有时两个开关都断开了，有时一个闭合而另一个断开，也有可能两个开关同时闭合。最后一种情况下，电路中电流的示意图如下图所示。



电路的公用部分没有电流通过。

我们通过公用线路把两个电路合成了一条电路。通过这种方式，把连通两栋房子所需的导线从 4 根减少到了 3 根，节约了 25%的导线。

如果整个系统必须铺设很长的线路，可能还得绞尽脑汁再削减一根导线以进一步节约开支。遗憾的是，对于眼前这套使用 1.5 伏干电池和小灯泡的系统来说，这是不可行的。但如果我们把它们换成 100 伏的电池和大得多的灯泡，问题就迎刃而解了。

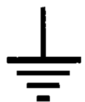
以下就是解决的思路：你不必非得用导线来完成电路的共用部分，你可以用其他的东西来代替导线。恰巧我们这儿有一个现成的大球，你可以用它来代替。这个大球直径近 7900 英里，由金属、岩石、水，以及有机质（其中大部分是没有生命的）组成。我们称这个巨大球体为“地球”。

上一章里，在讲导体的时候提到了银、铜和金，但没有提过岩石层和覆盖层。实际上，泥土并不是一个很好的导体，尽管有一些泥土（例如沼泽）的导电性比其他的（例如干沙）要好一些。但是我们知道导体有一条性质：截面越大导电性越好。一条很粗的导线，其导电性要远远好于一条细导线。这就是地球所拥有的优势。它实在是太大了。

要想用地球充当导体，可不是随便在西红柿地里插根线那么简单。你必须使用跟地球有充分接触的物体，也就是有很大表面积的导体。这儿有个不错的解决方案，即使用一个至少 8 英尺长、1/2 英寸粗的铜柱电极。它提供了与地球 150 平方英寸的接触面积。你可以用一个大锤子把这个电极砸进地里，然后在上面接上一根导线。或者，如果你家的水管是用铜做的，并且是从屋外的地下接过来的，那么你可以在水管上接上导线。

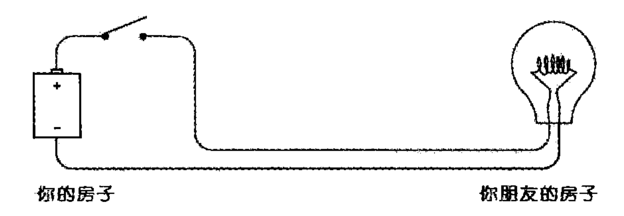
关于电流的接地，在英国人们称其为“earth”，在美国叫“ground”。“ground”这个词的含义有一些含混不清，因为它也经常用来表示我们前文中所说的“公用电路”。在本章中，除非特别指明，“ground”都表示物理接地。

在电路图中，人们用以下符号表示接地。

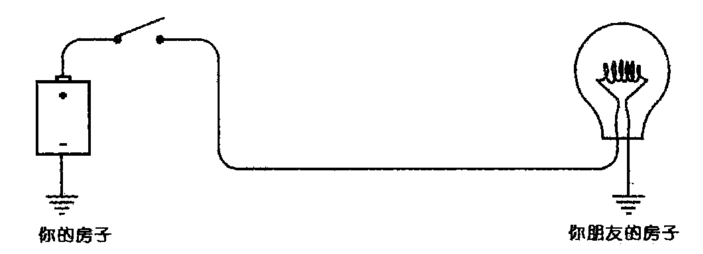


电气工程师们一般都会使用这个符号，因为他们不喜欢花时间去描画一个被埋在地下的8英尺长的铜柱。

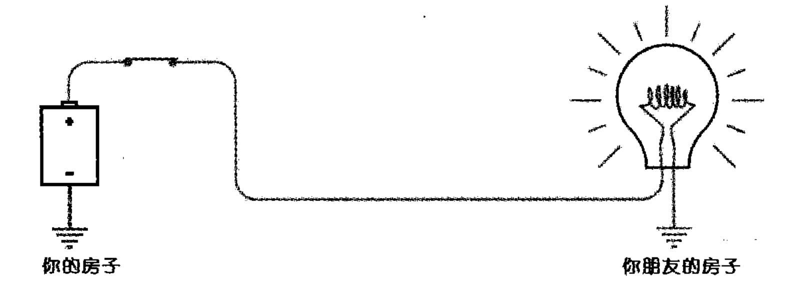
我们来看看新的电路是如何工作的。先从一个单向的电路开始分析。



如果你使用的是高压电池和大灯泡，你只需要在你和你朋友的房子之间接一根导线，因为可以把地球当成一条导线。



当你闭合开关时，电流就会按下图流动。



电子从你朋友房子的地下出发，经过灯泡、导线和你房间的开关，最后回到电池的正极。而电子最初是从电池的负极传入地下的。

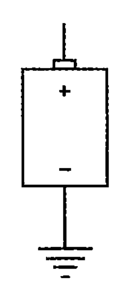
你或许还真的很想看到电子从埋在你家后院的 8 英尺长的铜柱进入地下，然后飞速通过大地到达埋在你朋友家后院的铜柱的情景。

然而每时每刻地球都在充当着全世界成千上万条电路的导线，想到这些，你可能会感到迷茫：电子怎么知道它要去什么地方？其实它们不知道。换一种描述地球的模式或许更恰当些。

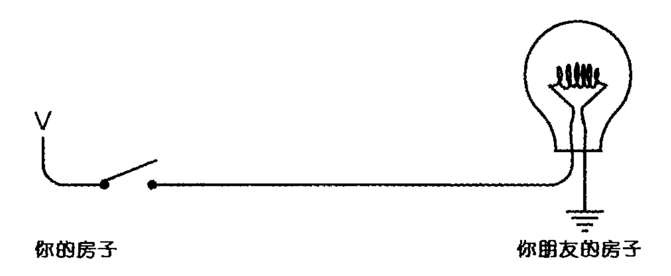
没错，地球是一个巨大的导体，但是我们也可以把它看做是电子的来源和储藏库。地球之于电子就恰如海洋之于水滴。地球是一个近乎无尽的电子之源，同时也是一个无比庞大的电子池。

不过地球还是有一定的电阻的。所以当我们使用 1.5 伏干电池和小灯泡时，不能通过接地来节约我们所需的线路开支。对于低电压电池而言，地球的电阻实在是太高了。

你可能注意到，前面两幅图中所画的电池，其负极都接地了，如下图所示。



以后，我将不再画这个图例了。我将用大写字母“V”（表示电压）来代替它。因此单向的灯泡电报系统的电路示意图就变成了如下这个样子。

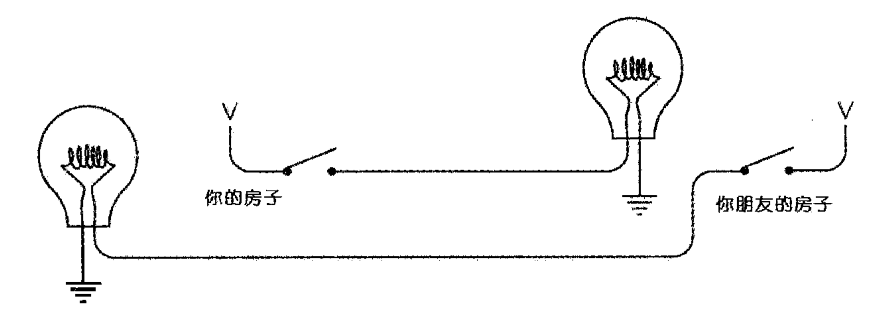


字母 V 是电压的意思，但是它也有吸尘器的意思。我们把 V 想象成一个电子吸尘器，然后把地面想象成电子的海洋。电子吸尘器通过电路把电子从地下拉出来，让它们沿设计好的线路开始工作（例如点亮灯泡）。

大地有时也被认为是零电势（zero potential）点。也就是说现在没有电压。电压——正如我之前介绍的——是电能做功的势，这与悬在空中的砖所潜藏的势能是类似的。零电势就像放置在地面上的砖一样——已经没有空间可以让它下落了。

在第 4 章中，我们最初注意到的几个问题之一便是电路是一条回路。不过我们的新电路看起来可不像条回路。实际上，它仍然是。你可以使用负极接地的电池来替换掉字母 V，然后把所有接地标志用线连起来。最后就会得到一张与本章开始处一模一样的电路图。

因此，在一对铜柱电极（或者自来水管）的帮助下，我们只用了两根导线，就冲破了你与你朋友家之间围墙的阻隔，构建了一个双向的莫尔斯码发送系统。



这条电路在功能上与之前的那个电路是完全一样的，而在之前那条电路中，我们使用了 3 条导线来穿越房子之间的围墙。

在本章中，我们已在通信的演变中迈出了重要一步。之前我们使用莫尔斯码交流时，必须要在视线直视的范围里，并且要保证在手电筒光线可以传播的距离之内。

但是使用导线，我们不仅可以构建出一个可以绕过拐角的、能够在视线之外的发报系统，而且我们自己再也无需受距离的限制。我们可以跨越成百上千英里来进行通信，只需要铺设足够长的线路即可。

不过还有个问题。尽管铜是很好的导电体，但是它也不是完美的。线路越长，它们的电阻也就越大。电阻越大，线路中的电流就越少。电流越少，灯泡就越暗。

那么到底我们可以做出多长的导线呢？这要依实际情况而定。假如我们现在使用的是原来的 4 条导线的双向电路，而没有使用接地和公用电路，并且还用手电筒电池和小

灯泡。为了节约成本，也许你先从无线电室（Radio Shack，美国出售电器的连锁店）购买了一些 20 号规格的电话线，花费为每 100 英尺 9.99 美元。电话线一般是用来把扬声器连接到立体声音响系统上的。它有两个接头，因而对我们的电报系统来说，也是个不错的选择。如果你和你朋友的卧室之间的距离小于 50 英尺，这卷电话线就足够了。

导线的粗细使用美国线径标准（American Wire Gauge，AWG）进行度量。AWG 数值越小，导线越粗，电阻也就越小。你买的 20 号规格的电话线，其直径大约是 0.032 英寸，大约每 1000 英尺有 10 欧的电阻，也就是说在你们卧室间的往返距离——100 英尺左右，大约有 1 欧的电阻。

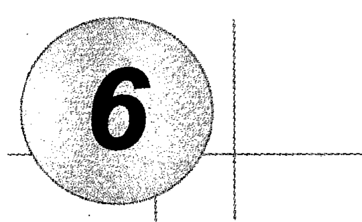
情况其实不错，但是如果我们需要铺设一英里的线路呢？导线的电阻将大于 100 欧。记得在上一章中提到过，我们的小灯泡电阻只有 4 欧姆。根据欧姆定律，可以很容易计算出电路中的电流已经不再是 0.75 安了（3 伏除以 4 欧），现在它将变得比 0.03 安还要小（3 伏除以 100 欧以上）。几乎可以肯定，这点电流不足以点亮灯泡。

使用粗一点的导线是一个不错的解决方案，但是那会比较昂贵。10 号规格线（无线电室用来作为汽车耦合线出售，其价格为每 35 英尺 11.99 美元，你需要两倍的长度，因为这种导线只有 1 个接口）大约有 0.1 英寸粗，不过它每 1000 英尺只有 1 欧的阻抗，也就是每英里 5 欧。

另一种解决方案是增加电压，使用更大电阻的灯泡。例如，一个能够照亮房间的 100 瓦灯泡，是设计成在 120 伏电压下工作的（美国市电电压），大约有 144 欧的电阻。导线的电阻对整个电路的影响将变得小许多。

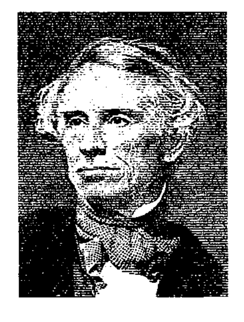
在 150 年前，人们在铺设第一个跨越美洲和欧洲的电报系统时，这些都是面临的问题。如果忽视了线路直径还有高电压的因素，电报线路将完全无法持续工作。根据设计方案，系统距离跨度的极限是 200 英里。这个长度与纽约和加利福尼亚间数千英里的距离相比，还是有很大差距的。

这个难题的解决方案——不是给手电筒的，而是给“滴滴答答”的近代电报系统的——尽管它只不过是个很简陋的装置，但是正是基于这个装置，整个计算机系统才被构建出来。



6 电报机与继电器

萨缪尔·芬利·布里斯·莫尔斯（Samuel Finley Breese Morse）1791 年出生于马萨诸塞州的查尔斯顿镇。这座小镇曾是邦克山战役打响的地方，位于波士顿的东北部。莫尔斯出生那年，距美国宪法制定刚过两年，而当时乔治·华盛顿正处在他的第一个总统任期内；叶卡捷琳娜大帝正统治着俄国；路易十六和玛莉·安托瓦内特将在 2 年后的法国大革命中人头落地。然后在 1791 年，莫扎特完成了他最后一部歌剧——《魔笛》(The Magic Flute)，不久后在 35 岁时与世长辞。



莫尔斯在耶鲁大学深造，并在伦敦学习艺术。后来他成为了一名成功的画家。他的作品《拉法耶特将军》(General Lafayette，1852 年)，现在还悬挂在纽约市政大厅。1836 年，他以独立候选人的身份参与竞选纽约市市长，并获得了 5.7%的选票。他还是一个早期的摄影爱好者。莫尔斯接受路易斯·达盖尔（Louis Daguerre）的亲自指点，学习银板照相术，并且拍摄了美国最早的一些银板照片。在 1840 年，他把这项技术传授给了 17 岁的马修·布雷迪（Mathew Brady），而布雷迪后来与同事一起，完成了一组非常有纪念意义的关于美国内战、亚伯拉罕·林肯（Abraham Lincoln），还有萨缪尔·莫尔斯本人等的摄影作品。

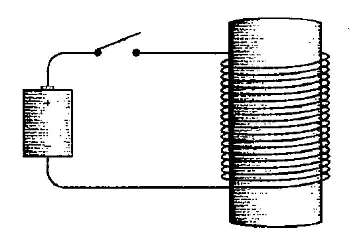
但是，这些只是他兼收并蓄职业生涯中的点缀而已。萨缪尔·莫尔斯之所以被现在的人们所熟知，还是因为他发明的电报机，以及以他名字命名的电码。

全球性即时通信对于我们来说已经司空见惯了，这项技术其实是在近代才得到发展的。回溯到19世纪早期，你也可以进行即时通信或者远距离通信，但是不能同时做到这两点。即时通信受声音传播距离的限制（没有扩音器可以用），或者受视野的限制（或许可以使用望远镜作为辅助工具）。使用信件可以进行更远距离的通信，但是寄信耗费的时间太多，并且需要马匹、火车或轮船。

莫尔斯的发明问世前的几十年里，人们为了提高远距离通信的速度，做过很多尝试。技术上比较简单的方法是，雇用一些人站在山上，作为中继系统，挥旗发出旗语。而在技术上稍复杂的解决方案是，使用带有机械臂的大型装置，代替人做挥旗的工作。

"电报"（telegraph，字面意思就是"远距离书写"）这个想法在19世纪早期就出现了，而且在萨缪尔·莫尔斯1832年开始实验之前，其他的发明家就已经开始研究它了。理论上说来，电报机的原理是很简单的：在线路的这一端采取一些措施，使线路的另一端发生某种变化。这恰好与我们上一章中用手电筒远距离发送信息所用到的方法很一致。不过呢，莫尔斯先生不会用灯泡作为他的信号发生装置了，因为最早的可使用的电灯泡到1879年才被发明出来。作为替代手段，莫尔斯利用了电磁（electromagnetism）现象。

如果你手头有一根铁棒，那么在上面用细导线绕几百圈，然后在导线上接通电流，铁棒就变成了一块磁铁。现在它可以吸引其他的铁块和钢块（电磁铁上缠绕足够多的细导线，会产生足够强的电阻，能防止电磁铁产生短路现象）。断开电流，铁棒将丧失磁性。



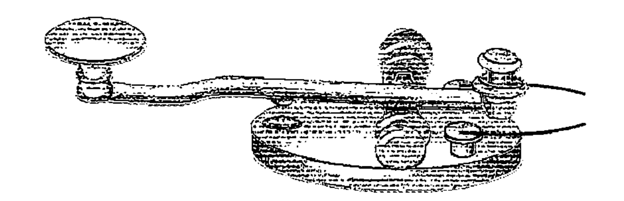
电磁铁是电报机的基础。在线路的一端闭合或断开开关，可以使线路另一端的电磁铁有所动作。

实际上，莫尔斯的第一个电报机比后来演化出的版本要复杂。莫尔斯认为，发报系

统应该确实能在纸上写出些什么东西，或者就像计算机用户后来描述的那样，“输出一份硬拷贝。”当然，电报机输出的不一定非得是单词，因为那样做就太过复杂了。但是不管是杂乱无章的线条还是点和划，在纸上总应该写些东西。注意，莫尔斯现在一头扎进了一个需要纸张和阅读的模型，这很像瓦伦丁·霍伊（Valentin Haüy）的观点——盲人用的书应该印有凸起的字母。

尽管萨缪尔·莫尔斯在 1836 年通知过专利局，他成功发明了电报机，但是直到 1843 年，他才说服美国国会，为其创建了一个公共基金。1844 年 5 月 24 日，这是历史性的一天，当华盛顿特区和马里兰州巴尔迪摩市之间的电报线路架设完成时，一条信息被成功地传递，内容是圣经中的句子：“What hath God wrought!”

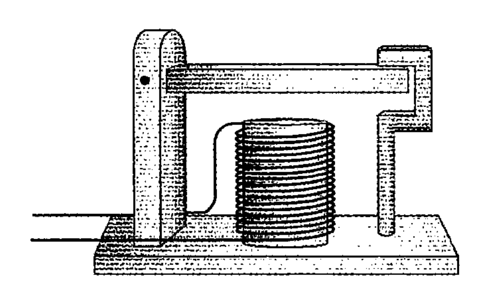
传统电报机中用来发送信息的电键，其外形如下图所示。



它虽然外表奇特，其实只是一个被设计成有“最大开闭速度”的开关而已。如果需要长时间使用电键，最舒适的方法是，用拇指，食指和中指握住手柄，轻击它使其上下移动。保持电键的按下状态一小段时间，就会产生一个“点”的莫尔斯码。按下状态保持的时间更长一些就会产生一个“划”的莫尔斯码。

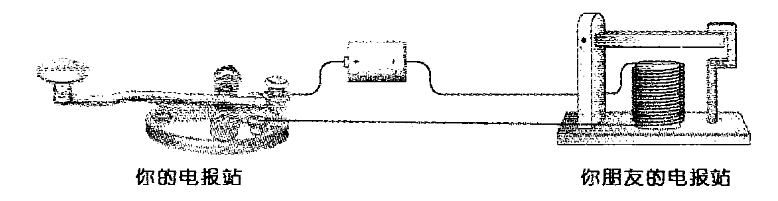
线路的另一端是一个接收器，它主要是由一块电磁铁拉动一根金属杆构成的。最初，电磁铁控制的是一只钢笔。有一个装置通过使用一个压紧的弹簧来拉动一卷纸经过设备，与电磁铁连接着的钢笔就会弹起或落下，在纸上画出点和划。能读懂莫尔斯电码的人员就可以把这些“点”和“划”译成字母和单词了。

当然，我们人类是个懒惰的物种，电报操作员们很快发现，他们可以很容易地通过听钢笔弹起和落下的声音来翻译电码。在传统电报机中的“发声器”的帮助下，钢笔最终被废弃，整个装置看起来如下图所示。



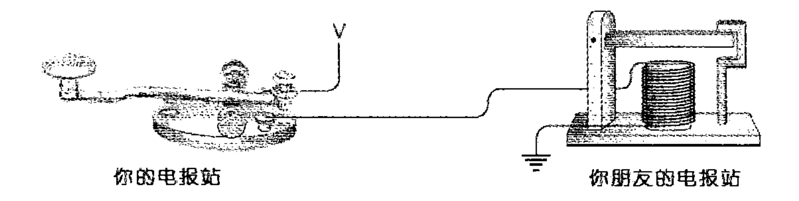
当电报机的电键被按下时，发声器中的电磁铁拉动上面的活动横杠下降，它会发出“滴”的声音。当松开电键的时候，横杠弹回到原来的位置，发出“嗒”的声音。一次快速的“滴-嗒”声代表点；一次慢速的“滴——嗒”声则代表划。

电键、发声器、电池，还有一些导线连接到一起，电报机的电路示意图与前面章节中的手电筒电路图很相似。



正如我们以前所发现的那样，不必非得用两根导线来连接两个电报站。如果地球能为我们提供电路的另一半的话，一根导线就足够了。

就像我们在上一章所做的那样，我们可以把接地的电池用大写“V”表示。因此完整的单向系统如下图所示。



双向通信仅仅需要再增加一个电键和发报人。这与我们前一章中的做法是类似的。

电报机的发明标志着现代通信的开始。人们第一次能够在视线或者听力之外的距离范围进行实时交流了，而且信息传递的速度比骏马疾驰还要快。而更加耐人寻味的是，这个发明使用了二进制码。但是在后来的电子和无线通信（包括电话、无线电、电视）所使用的通信模式中，二进制码被废弃了，直到后来它又被应用在了电脑、光盘、数字

影碟、数字卫星电视广播和高清电视上。

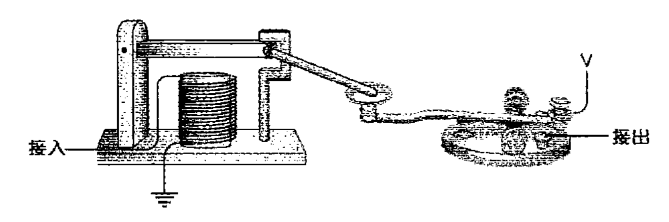
莫尔斯的电报机在某种程度上要胜过其他的设计，因为它对线路没有苛刻的要求。当你连好电键和发声器之间的线路后，它们一般都可以正常工作。而其他的电报系统对线路的要求就没这么简单了。但是正如我上一章提到的那样，这种电报机最大的问题就是长导线所带来的电阻。尽管一些电报线路使用高达 300 伏的电压，而使有效距离能够超过 300 英里，但是线路还是不能无限延长。

显然，设置一个中继系统是解决该问题的一个方案。每隔 200 英里左右，为一个工作人员装配好发声器和电键，他就可以接收信息，然后再把它转发出去。

现在，想象你已经被电报公司聘用，成为中继系统的一部分。他们把你扔在纽约和加利福尼亚之间的一个无名之地，让你在一个只有一桌一椅的小屋里工作。一条导线从东面的窗户伸进来，连接到发声器上。而你的电报机电键连到电池上，最后线路从西窗伸出去。你的职责就是接收从纽约发来的信息，然后转发它们，最终使它们到达加利福尼亚。

开始时，你喜欢接收完一条完整的信息后再把它转发。首先，根据发声器发出的滴答声，将字母记下来；当信息接收完毕时，再开始用你的电键来发送。最后，你终于掌握了诀窍，在听到滴答声的同时就可以发送信息，不需要再把信息记录下来了。这节约了不少时间。

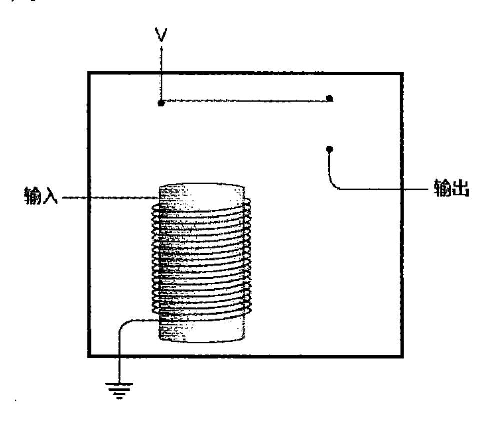
有一天，当你正在转发信息时，瞄了瞄发声器上那根上蹿下跳的横杠，又看了看在电键上上下翻飞的手指。然后你就这样来来回回地瞅来瞅去，恍然发现发声器上下跳跃的节奏与电键是一致的。因此你就去外面找了根小木棍，然后用木棍和一些细绳把发声器和电键连接到了一起，如下图所示。



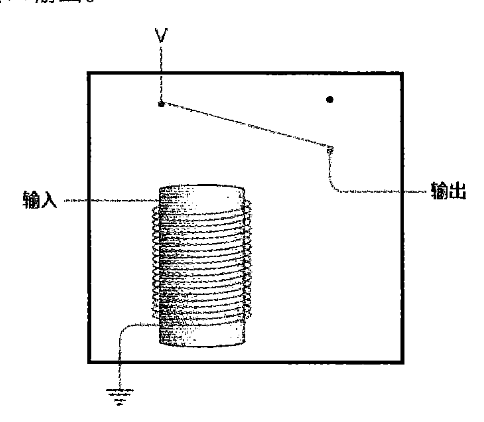
现在，设备可以自己工作了，而下午剩余的时间嘛，你就可以去休个假，钓个鱼。

这真是个有趣的想法，但是事实上，萨缪尔·莫尔斯在早些时候就已经领会了这个设备的概念。刚刚我们发明的这个设备称做“继电器”。继电器与发声器很像，传进来的电流驱动电磁铁拉动金属杠，金属杠同时又作为一个开关的组成部分，而这个开关连接着电池和输出线路。通过这种方法，输入的比较弱的电流就被“放大”成了较强的输出电流。

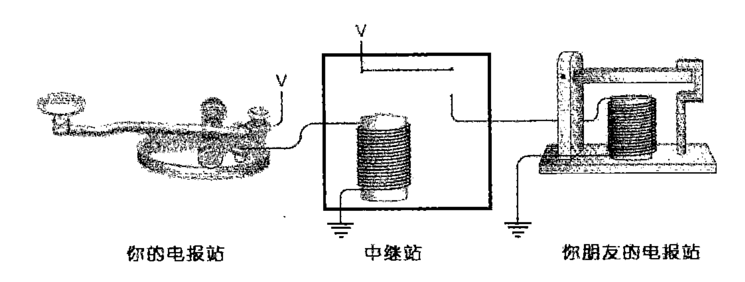
继电器的示意图如下。



当输入的电流触发了电磁铁，电磁铁把一个弹性金属条吸附下来，就像闭合了开关一样，使电流可以从接口输出。

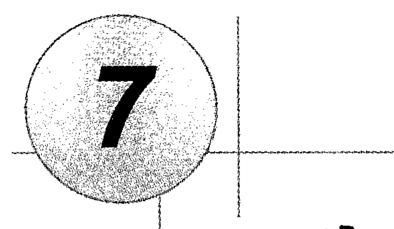


因此，把一个电报机电键、一个继电器，还有一个发声器连接好，差不多就是下图的这个样子。



继电器是一个意义非凡的设备。当然，它是一个开关，但是这个开关的闭合和断开并不是由人来操纵的，而是由电流控制的。你可以通过它来完成一些不可思议的事情。实际上，使用它，你甚至可以装配好一台近乎完整的计算机来！

不错，继电器是一项如此出色的发明，以至于电报博物馆也为它留有一席之地。让我们悄悄拿起一台，藏在外套里，飞快地从警卫身边溜过去。这台继电器用起来将非常方便。不过我们在使用它之前，还要学会如何计数。



我们的十个数字

人们很容易理解，语言只不过是一种编码。我们之中的许多人在学校里至少都学过一门外语。所以我们知道，英文中的“cat”（猫）在其他语言中可以写做 gato、chat、Katze、KOIIIK 或 *kátta*。

然而，数字似乎并不是那么容易随文化的不同而改变。不论我们说什么语言，或对数字使用什么样的发音，在这个星球上几乎所有人都用以下方式来书写数字：

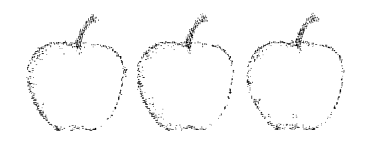
1　2　3　4　5　6　7　8　9　10

这难道不就是数学被称做“通用语言”的理由么？

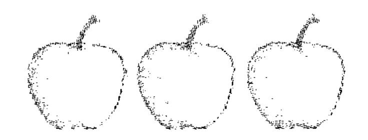
数字当然是我们平常所能接触到的一种最抽象的编码。当我们看到数字：

3

不需要立刻将它与任何事物联系起来。我们可能会联想到 3 个苹果或者 3 个别的什么东西。但是当我们从上下文中得知该数字表示的是某个小孩的生日、电视频道、曲棍球赛的得分或蛋糕食谱中面粉的杯数时，也能够像认为它代表 3 个苹果时一样自然。因为数字最开始产生时就很抽象，所以对于我们来说，理解这样一个问题会有一点困难。这个问题就是如下数量的苹果：



并不一定要用符号“3”来表示。本章及下一章中的很大一部分内容将专门用来说明如下这些苹果：



同样可以用“11”来表示。

首先让我们遗忘数字 10 原有的那些特性。大多数文明都是建立在以 10 为基数的数字系统上的（有的时候是以 5 为基数），这种情况并不奇怪。最开始，人们用自己的手指来计数。如果我们人类有 8 个或 12 个手指，那么我们的计数方式就会和现在有所不同。英语中 Digit（数字）这个词同时也有手指、脚趾的意思，并且还有数字的意思，这并不是巧合。而 five（五）和 fist（拳头）这两个单词的拥有相同的词根也是同样的道理。

在这个意义上，以 10 为基数或使用十进制数字系统完全是随意的。而且，英文中还对基于十的数字赋予了几乎神奇的意义，并且给了它们特有的名字：十个一年是一个十年（decade）；十个十年是一个世纪（century）；十个世纪就是一个千年（millennium）。一千个一千就是一个百万（million）；一千个百万就是一个十亿（billion）。以下都是 10 的各次幂。

$10^1 = 10$

$10^2 = 100$

$10^3 = 1000$（千）

$10^4 = 10,000$

$10^5 = 100,000$

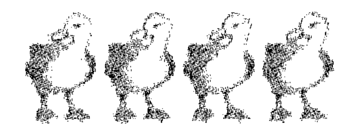
$10^6 = 1,000,000$（百万）

$10^7 = 10,000,000$

$10^8 = 100,000,000$

$10^9 = 1,000,000,000$（十亿）

大多数历史学家认为数字最初起源于对事物的计数，例如：人数、财产或商业交易的计数等。举个例子，如果有一个人有四只鸭子，用图画表示为：



后来，专门负责画鸭子的这个人会想："为什么我非得要画四只鸭子？为什么我不画一只鸭子再用划线或其他事物来表示有四只鸭子呢？"



然后直到有一天，出现了一个人，他拥有 27 只鸭子，这种划线的方法就显得很可笑了。



有人说："必须想一种更好的方法。"于是一个数字系统就诞生了。

所有早期的数字系统中，只有罗马数字沿用到了今天。我们可以在表盘上、纪念碑和雕像的日期上、一些书的页码中，或者在条款的概述中看到罗马数字，而令人最烦恼的就是电影的版权声明（必须足够快地破译位于演职人员表末尾的"MCMLIII"才能知道这部影片是哪一年发行的）。

27 只鸭子用罗马数字表示为：



这个概念很容易理解：X 表示 10 个划线，V 表示 5 个划线。

沿用到今天的罗马数字符号有：

I V X L C D M

这里，字母 I 表示 1，可以看做是一个划线或者一根伸出的手指。字母 V 像一只手，表示 5。两个 V 是一个 X，代表数字 10。L 是 50。C 来自单词 centum，表示 100。D 是 500。最后一个，M 来自于拉丁文 mille，意为 1000。

尽管我们可能不会认同，但在很长一段时间内，罗马数字被人们看做是易于加减的，这也是为什么罗马数字在欧洲作记账之用一直沿用到今天。实际上，两个罗马数字相加的时候只不过是利用几个规则将两个数合并，这个规则是：五个 I 是一个 V，两个 V 是一个 X，五个 X 是一个 L，以此类推。

但是用罗马数字进行乘法和除法却很复杂。很多其他早期数字系统（像古希腊数字系统）和罗马数字系统相似，它们在用于复杂运算方面同样也存在一定的不足。尽管古希腊人发明的非凡的几何学至今仍然是高中生的一门课程，但古希腊人并不是以代数而著称的。

如今我们所用的数字系统通常被称为阿拉伯数字，也可以称为印度-阿拉伯数字系统。它起源于印度，被阿拉伯数学家带入欧洲。其中最著名的就是波斯数学家穆罕默德·伊本穆萨·奥瑞兹穆（根据这个人的名字衍生出英文单词“algorithm”，算法），他在公元 825 年左右写了一本关于代数学的书，其中就用到了印度的计数系统。其拉丁文译本可追溯到公元 1120 年，它对加速整个欧洲从罗马数字到阿拉伯数字系统的转变有着重要影响。

阿拉伯数字系统不同于先前的数字系统，体现在以下三点。

- 阿拉伯数字系统是和位置相关的。也就是说，一个数字的位置不同，其代表数量也不同。对于一个数而言，其数字的位置和数字的大小一样，都是很重要的（但实际上，数字的位置更重要）。100 和 1,000,000 这两个数中都只有一个 1，而我们知道，1,000,000 要远远大于 100。
- 实际上在早期的数字系统中也有一点是阿拉伯数字系统所没有的，那就是用来表示数字 10 的专门的符号。而在我们现在使用的数字系统中是没有代表 10 的专门符号的。
- 另一方面，实际上阿拉伯数字也有一点是几乎所有早期数字系统所没有的，而这恰恰是一个比代表数字 10 的符号还重要得多的符号，那就是 0。

是的，就是 0。小小的一个零无疑是数字和数学史上最重要的发明之一。它支持位置

计数法，因此可以将 25、205 和 250 区分开来。0 也简化了与位置无关的数字系统中的一些非常复杂的运算，尤其是乘法和除法。

阿拉伯数字的整体结构可以以我们读数字的方式来展现。以 4825 为例，我们读做“四千八百二十五”，意思就是：

四千
八百
二十
五

或者，我们也可以将此结构以如下写法写出：

$$4825 = 4000 + 800 + 20 + 5$$

或者，对其进一步分解，可以将数字写做：

$$\begin{aligned}4825 = 4 \times 1000 &+ \\ 8 \times 100 &+ \\ 2 \times 10 &+ \\ 5 \times 1&\end{aligned}$$

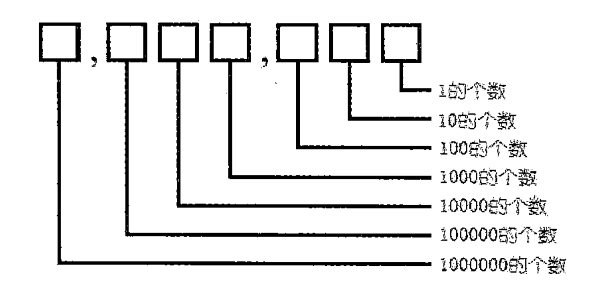
或者，以 10 的整数次幂的形式来表示：

$$\begin{aligned}4825 = 4 \times 10^3 &+ \\ 8 \times 10^2 &+ \\ 2 \times 10^1 &+ \\ 5 \times 10^0&\end{aligned}$$

记住任何数的 0 次幂都等于 1。

一个多位数中的每一位都有其各自特定的意义，如下图所示。这 7 个方格能代表 0～9,999,999 中的任何一个数字。

每个位置代表 10 的一个整数次幂。我们不需要一个专门的符号来表示数字“10”，因为我们可以将 1 放在不同的位置，并用 0 作为占位符。



另一个好处就是，以同样的方式将数字置于小数点右边可以表示分数。数字42,705.684就是：

$$4 \times 10,000 +$$
$$2 \times 1000 +$$
$$7 \times 100 +$$
$$0 \times 10 +$$
$$5 \times 1 +$$
$$6 \div 10 +$$
$$8 \div 100 +$$
$$4 \div 1000$$

这个数也可以写为不含除法的形式，如下：

$$4 \times 10,000 +$$
$$2 \times 1000 +$$
$$7 \times 100 +$$
$$0 \times 10 +$$
$$5 \times 1 +$$
$$6 \times 0.1 +$$
$$8 \times 0.01 +$$
$$4 \times 0.001$$

或用10的幂的形式来表示：

$$4 \times 10^4 +$$
$$2 \times 10^3 +$$
$$7 \times 10^2 +$$
$$0 \times 10^1 +$$
$$5 \times 10^0 +$$

$6 \times 10^{-1} +$
$8 \times 10^{-2} +$
4×10^{-3}

注意，10 的幂指数是如何减小到 0 再变为负数的。

我们知道，3 加 4 等于 7。类似地，30 加 40 等于 70，300 加 400 等于 700，3000 加 4000 等于 7000。这就是阿拉伯数字的“闪光”之处。任何长度的十进制数相加时，只要根据一种方法将问题分成几步即可，而每个步骤最多只是将两个一位数字相加而已。这就是为什么以前有人会强迫你记住加法表的原因。

+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

从最上边的一行和最左边的一列分别找出要相加的两个数字，这一行与这一列的交叉点就是所要得到的和。例如，4 加 6 等于 10。

同样，当你想将两个十进制数相乘的时候，方法可能稍微复杂些，但是你仍然只需要将问题分解成几步，做加法和一位数的乘法即可。在你的小学时代你一定也被要求必须记住下面的乘法表。

×	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45

续表

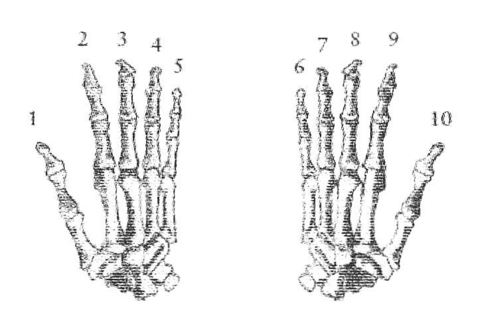
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

位置计数系统的好处并不在于它有多么好用，而在于对非十进制的系统而言，它仍然是易于实现计数的。我们现有的计数系统并不适用于每种情况。以 10 为基数的数字系统最大的问题是它对于卡通人物没有任何意义。大多数卡通人物每只手（或爪子）只有 4 根手指，因此它们需要一个以 8 为基数的计数系统。而有意思的是，许多我们在十进制数中所了解到的知识同样适合卡通朋友们所钟爱的八进制计数系统。

8

十的替代品

对于我们人类而言，10 是一个非常重要的数字。10 是我们大多数人拥有的手指或脚趾的数目，当然我们也希望所有人的手指和脚趾都是 10 个。因为手指非常便于计数，于是我们人类已经适应了这个以 10 为基数的数字系统。



就如前面章节中所提到的，我们现在所用的数字系统是基于 10 的数字系统的，或称为十进制。我们已经非常习惯这个数字系统了，因此起初也很难再构想出其他的数字系统。的确，当我们看到数字 10 的时候，我们不自觉地就会想到这个数字代表了下面这么多只鸭子：

10 =

但是，数字10之所以指的是这么多只鸭子，其唯一理由就在于这些鸭子的数目与我们的手指数目相同。如果人类的手指不是10根，我们数数的方式就会有所不同，那么10就会是另外一个含义。同样，数字10可以代表这么多只鸭子：

10 =

或者是这么多只的鸭子：

10 =

甚至可以是这么多只的鸭子：

10 =

当我们认识到，10 可以表示两只鸭子，我们就可以解释开关、导线、灯泡和继电器（进一步推广到计算机）是如何表示数字的了。

如果人类像卡通人物那样每只手有 4 根手指会怎样呢？我们可能就不会想到建立一个以10为基数的数字系统。相反我们会自然而然地、不可避免地想到建立一个以8为基数的数字系统。我们就不会称这个系统为十进制数字系统，而称之为八进制数字系统。

如果我们的数字系统是以8为基础而建立的，我们就不需要如下这个符号：

9

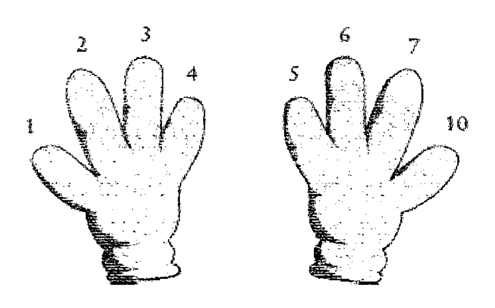
将这个符号给卡通人物看的话，你会得到这样的反应："这是什么？它代表什么？"如果继续仔细考虑一下，你会发现我们连这样的一个符号也不会需要：

8

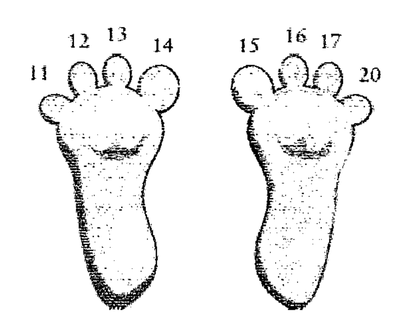
在十进制中，10没有特定的符号，因此在八进制中，同样也没有表示8的特定符号。

十进制中我们的计数方式是：0、1、2、3、4、5、6、7、8、9，然后是 10。在八进制中数字系统中计数方式是：0、1、2、3、4、5、6、7，而后是什么呢？我们已经将符

号用完了。在这里唯一有意义的只有10，而在这里的正确答案恰恰就是10。在八进制中，7之后的下一个数字是10。但是这个“10”代表的并不是人类手指的数量。在八进制中，“10”代表的是卡通人物手指的数量。



如果继续用脚趾数下去的话，就是这样：



使用非十进制的数字系统时，你可以将“10”读作“一零”，这样可以避免一些混淆。类似地，“13”读作“一三”，“20”读作“二零”。要想真正避免混淆，可以将“20”读作“基于8的数二零”或“八进制二零”。

尽管已经用完了所有的手指和脚趾，但我们仍能够用八进制继续数下去。这与十进制基本相同，但是我们跳过了数字8和9。当然，相同数字所代表的数量是不同的：

0，1，2，3，4，5，6，7，10，11，12，13，14，15，16，17，20，21，22，23，24，25，26，27，30，31，32，33，34，35，36，37，40，41，42，43，44，45，46，47，50，51，52，53，54，55，56，57，60，61，62，63，64，65，66，67，70，71，72，73，74，75，76，77，100……

这里最后一个数字我们读做“一零零”。这是卡通人物手指的数量自身相乘所得的结果。

当书写十进制和八进制数时，我们可以利用下标标注来区别不同数字系统，避免混淆。下标 TEN 表示十进制，EIGHT 表示八进制。

这样，白雪公主遇到的小矮人数量是 7_{TEN} 或 7_{EIGHT}；

卡通人物的手指数是 8_{TEN} 或 10_{EIGHT}；

贝多芬所写的交响曲的数目是 9_{TEN} 或 11_{EIGHT}；

人类手指的数量是 10_{TEN} 或 12_{EIGHT}；

一年中的月份数是 12_{TEN} 或 14_{EIGHT}；

两个星期的天数是 14_{TEN} 或 16_{EIGHT}；

“情人”的生日庆祝会是 16_{TEN} 或 20_{EIGHT}；

一天中的小时数是 24_{TEN} 或 30_{EIGHT}；

拉丁字母表中的字母数是 26_{TEN} 或 32_{EIGHT}；

一夸脱的液体相当于的盎司数是 32_{TEN} 或 40_{EIGHT}；

一副纸牌的张数是 52_{TEN} 或 64_{EIGHT}；

棋盘的格数是 64_{TEN} 或 100_{EIGHT}；

日落大道上最有名的地址是 77_{TEN} 或 115_{EIGHT}；

美式足球场地的面积是 100_{TEN} 或 144_{EIGHT}；

温网首届女子单打的初赛人数是 128_{TEN} 或 200_{EIGHT}；

孟菲斯市的面积是 256_{TEN} 或 400_{EIGHT}。

注意，在上面一系列的八进制数中出现了几个好整数（nice round number），如 100_{EIGHT}、200_{EIGHT} 和 400_{EIGHT}。根据规定，在十进制中，好整数通常是指结尾有若干个零的数。在结尾有两个零的十进制数代表的是 100_{TEN}，而 100_{TEN} 表示 10_{TEN} 乘以 10_{TEN}。在八进制数中，结尾有两个零代表是 100_{EIGHT}，而 100_{EIGHT} 表示 10_{EIGHT} 乘以 10_{EIGHT}（或 8_{TEN} 乘以 8_{TEN}，即 64_{TEN}）。

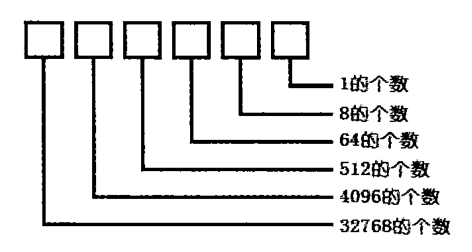
你可能还会注意到，这些好整数 100_{EIGHT}、200_{EIGHT} 和 400_{EIGHT} 在十进制中分别与 64_{TEN}、128_{TEN} 和 256_{TEN} 相等，它们全是 2 的整数次幂。这是非常有意义的。例如，400_{EIGHT} 等于 4_{EIGHT} 乘以 10_{EIGHT} 乘以 10_{EIGHT}，而这里所有的数都是 2 的整数次幂。任何 2 的整数次幂乘以 2 的整数次幂的结果依然是 2 的整数次幂。

下表给出了一些 2 的整数次幂的十进制及其对应的八进制的表示形式。

2 的整数次幂	十进制	八进制
2^0	1	1
2^1	2	2
2^2	4	4
2^3	8	10
2^4	16	20
2^5	32	40
2^6	64	100
2^7	128	200
2^8	256	400
2^9	512	1000
2^{10}	1024	2000
2^{11}	2048	4000
2^{12}	4096	10000

最右列的好整数暗示我们，十进制以外的数字系统可能对使用二进制码有所帮助。

在结构上，八进制数字系统与十进制数字系统并没有什么不同。它们只是在细节上存在一些差异。例如，八进制数中的每个位所代表的值是该位数字乘以 8 的整数次幂的结果。



这样，一个八进制数 3725_{EIGHT} 可以分解成如下形式：

$$3725_{EIGHT}=3000_{EIGHT}+700_{EIGHT}+20_{EIGHT}+5_{EIGHT}$$

同样还可以写成若干种不同的形式，下面是利用十进制的 8 的整数次幂写出的一种形式：

$$\begin{aligned}3725_{\text{EIGHT}} = & 3 \times 512_{\text{TEN}} + \\ & 7 \times 64_{\text{TEN}} + \\ & 2 \times 8_{\text{TEN}} + \\ & 5 \times 1\end{aligned}$$

采用八进制形式的 8 的整数次幂，表现形式如下：

$$\begin{aligned}3725_{\text{EIGHT}} = & 3 \times 1000_{\text{EIGHT}} + \\ & 7 \times 100_{\text{EIGHT}} + \\ & 2 \times 10_{\text{EIGHT}} + \\ & 5 \times 1\end{aligned}$$

还有一种拆分形式：

$$\begin{aligned}3725_{\text{EIGHT}} = & 3 \times 8^3 + \\ & 7 \times 8^2 + \\ & 2 \times 8^1 + \\ & 5 \times 8^0\end{aligned}$$

如果用十进制数计算出结果，会得到 2005_{TEN}。这就是将八进制数转化为十进制数的方法。

我们可以像进行十进制数的加法和乘法那样，对八进制数进行加法或乘法运算。不同的是，我们要采用不同的表来对各个数位进行乘法或加法运算。下面是八进制数的加法表。

+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

例如，$5_{EIGHT}+7_{EIGHT}=14_{EIGHT}$。我们还可以将两个更长的八进制数按照与十进制相同的方法相加：

$$\begin{array}{r} 135 \\ +\,643 \\ \hline 1000 \end{array}$$

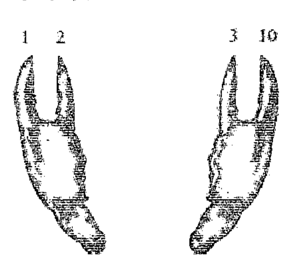
首先要从最右的一列开始，5 加 3 等于 10，结果为 0，进位 1；1 加 3 加 4 等于 10，结果为 0，进位 1；1 加 1 加 6 等于 10。

同样，在八进制中，2 乘以 2 结果依然为 4。但是 3 乘以 3 却不等于 9。那是多少呢？在这里，3 乘以 3 结果是 11_{EIGHT}，其与 9_{TEN} 所代表的数量相等。下面为八进制乘法的乘法表。

×	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61

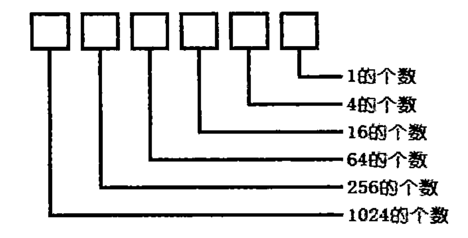
在这里，4 × 6 等于 30_{EIGHT}，而 30_{EIGHT} 与 24_{TEN} 等价，也就是与十进制中的 4 × 6 相等。

八进制数字系统与十进制数字系统一样，都是有效的。但是让我们更进一步来看，既然已经为卡通人物开发了一套数字系统，就让我们再制订一套适合龙虾的数字系统吧。实际上龙虾根本没有手指，但是在它们前爪的末端都有螯。适用于龙虾的数字系统是以 4 为基数的四进制（quaternary）数字系统。



四进制数字系统是像这样计数的：0，1，2，3，10，11，12，13，20，21，22，23，30，31，32，33，100，101，102，103，110，等等。

这里我不打算在四进制数上花太多的时间，因为接下来我们即将转入一个更为重要的话题。通过下图我们可以看出四进制中每一位是如何跟 4 的某个整数次幂相对应的。



四进制数 31232 可以写做如下形式：

$$
\begin{aligned}
31232_{\text{FOUR}} = {} & 3 \times 256_{\text{TEN}} + \\
& 1 \times 64_{\text{TEN}} + \\
& 2 \times 16_{\text{TEN}} + \\
& 3 \times 4_{\text{TEN}} + \\
& 2 \times 1
\end{aligned}
$$

等价于：

$$
\begin{aligned}
31232_{\text{FOUR}} = {} & 3 \times 10000_{\text{FOUR}} + \\
& 1 \times 1000_{\text{FOUR}} + \\
& 2 \times 100_{\text{FOUR}} + \\
& 3 \times 10_{\text{FOUR}} + \\
& 2 \times 1
\end{aligned}
$$

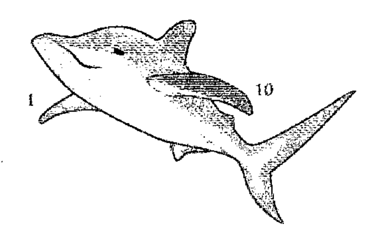
同样可以写成：

$$
\begin{aligned}
31232_{\text{FOUR}} = {} & 3 \times 4^4 + \\
& 1 \times 4^3 + \\
& 2 \times 4^2 + \\
& 3 \times 4^1 + \\
& 2 \times 4^0
\end{aligned}
$$

如果以十进制数的形式计算其结果，我们会发现 31232_{FOUR} 等价于 878_{TEN}。

现在，我们要做一个跳跃并且是最远的一跳。如果我们是海豚，那么就必须用两个鳍来计数。这个数字系统称为以 2 为基数的数字系统，或二进制。这样的话我们似乎只有两个数字了，这两个数分别为 0 和 1。

0 和 1 现在已不是我们要处理的全部问题了，而我们还需要练习一下才能习惯使用二进制数。二进制数最大的问题是数字用完得很快。例如，下图就是海豚如何用它的鳍来计数的例子。



是的，在二进制中，1 的下一个数字是 10。这让人惊讶，但也并不是什么意外。无论使用哪种计数系统，当单个的数字用完时，第一个两位数就是 10。二进制系统这样计数的：

0，1，10，11，100，101，110，111，1000，1001，1010，1011，1100，1101，1110，1111，10000，10001……

这些数看起来似乎很大，但实际上并不是这样的。准确地说，二进制数的长度增长得很快而非数值增大得快：

人类的头的个数是 1_{TEN} 或 1_{TWO}；

海豚身上鳍的个数是 2_{TEN} 或 10_{TWO}；

一个大汤匙中的小茶匙的数目为 3_{TEN} 或 11_{TWO}；

正方形的边数是 4_{TEN} 或 100_{TWO}；

人类一只手的手指数是 5_{TEN} 或 101_{TWO}；

昆虫的腿数是 6_{TEN} 和 110_{TWO}；

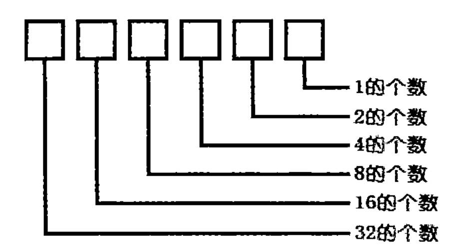
一周中的天数是 7_{TEN} 或 111_{TWO}；

八重奏中演奏者的人数是 8_{TEN} 或 1000_{TWO}；

银河系中行星（包括冥王星）的总数是 9_{TEN} 或 1001_{TWO}；

牛仔帽的重量以加仑计算为 10_{TEN} 或 1010_{TWO}。

在一个多位二进制数中，数字的位置和 2 的整数次幂的对应关系为：



因此，任何一个以 1 开头而后面全是 0 的二进制数一定都是 2 的整数次幂。幂指数就等于这个二进制数中 0 的个数。

以下是 2 的各整数次幂的扩展表，它可以用来说明这条规则。

2 的整数次幂	十进制数	八进制数	四进制数	二进制数
2^0	1	1	1	1
2^1	2	2	2	10
2^2	4	4	10	100
2^3	8	10	20	1000
2^4	16	20	100	10000
2^5	32	40	200	100000
2^6	64	100	1000	1000000
2^7	128	200	2000	10000000
2^8	256	400	10000	100000000
2^9	512	1000	20000	1000000000
2^{10}	1024	2000	100000	10000000000
2^{11}	2048	4000	200000	100000000000
2^{12}	4096	10000	1000000	1000000000000

假定有一个二进制数 101101011010，它可以写成：

$$
\begin{aligned}
101101011010_{TWO} = {} & 1 \times 2048_{TEN} + \\
& 0 \times 1024_{TEN} + \\
& 1 \times 512_{TEN} + \\
& 1 \times 256_{TEN} + \\
& 0 \times 128_{TEN} + \\
& 1 \times 64_{TEN} + \\
& 0 \times 32_{TEN} + \\
& 1 \times 16_{TEN} + \\
& 1 \times 8_{TEN} + \\
& 0 \times 4_{TEN} + \\
& 1 \times 2_{TEN} + \\
& 0 \times 1_{TEN}
\end{aligned}
$$

同样也可以用下面这种方式表示：

$$
\begin{aligned}
101101011010_{TWO} = {} & 1 \times 2^{11} + \\
& 0 \times 2^{10} + \\
& 1 \times 2^{9} + \\
& 1 \times 2^{8} + \\
& 0 \times 2^{7} + \\
& 1 \times 2^{6} + \\
& 0 \times 2^{5} + \\
& 1 \times 2^{4} + \\
& 1 \times 2^{3} + \\
& 0 \times 2^{2} + \\
& 1 \times 2^{1} + \\
& 0 \times 2^{0}
\end{aligned}
$$

如果将各个部分以十进制数的形式相加，会得到 2048 + 512 + 256 + 64 + 16 + 8 + 2，即 $2{,}906_{TEN}$。

为了更简明、更便捷地将二进制数转换为十进制数，你可能会更喜欢借助我准备的

模板来进行转换：

□	□	□	□	□	□	□	□
×128	×64	×32	×16	×8	×4	×2	×1

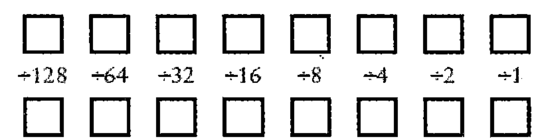
□+□+□+□+□+□+□+□=□

这个模板可以将一个数转换最大长度为 8 的二进制数，但是它很容易扩展。在使用模板的时候，将 8 位二进制数填入到上面一行的格子里，每个格子一位。做 8 个乘法运算，然后将结果填入到下面一行的 8 个小格子里。再将这 8 个格子中的数相加就会得到最终结果。下面就举例说明如何得到与二进制数 10010110 相等的十进制数。

1	0	0	1	0	1	1	0
×128	×64	×32	×16	×8	×4	×2	×1

128 + 0 + 0 + 16 + 0 + 4 + 2 + 0 = 150

将十进制数转换成二进制数就没那么直接了。下面是一个能够帮你将 0～255 范围内的十进制数转换为二进制数的模板。



这个转换实际上比看上去要麻烦得多，因此一定要仔细地按照下面的指导来操作。将整个十进制数（小于或等于 255）填入到上面一行最左端的格子中。用第一个除数（128）去除这个数。所得的商填入正下方的格子（左下角的格子），余数填入右边的格子（上面一行左数第二个格子）。用第一个余数再除以下一个除数 64。依照模板的顺序用同样的方法继续进行下去。

要记住，每次求得的商只能是 0 或者 1。如果被除数小于除数，商为 0，余数就是被除数。如果被除数大于或等于除数，那么商为 1，余数就是被除数减去除数所得之差。下面以 150 为例进行转换。

150	22	22	22	6	6	2	0
÷128	÷64	÷32	÷16	÷8	÷4	÷2	÷1
1	0	0	1	0	1	1	0

如果需要对两个二进制数进行加法或乘法，直接运算可能会比转换成十进制再进行

运算要简单些。这将是你真正喜欢二进制数的地方。想象一下，如果你记住了如下表格，掌握加法将是一件多么迅速的事情啊。

+	0	1
0	0	1
1	1	10

让我们用这个表格来计算两个二进制数的和：

```
  1100101
 +0110110
 --------
 10011011
```

从最右一列开始：1 加 0 等于 1。右边第二列：0 加 1 等于 1。第三列：1 加 1 等于 0，进 1。第四列：1（进位）加 0 加 0 等于 1。第五列：0 加 1 等于 1。第六列：1 加 1 等于 0，进 1。第七列：1（进位）加 1 加 0 等于 10。

乘法表甚至比加法表还要简单，因为该表可以由两个基本的乘法规则推导出来：任何数乘以 0 结果都为 0，任何数乘以 1，结果都是这个数本身。

×	0	1
0	0	0
1	0	1

以下是 13_{TEN} 与 11_{TEN} 的二进制乘法运算过程：

```
      1101
    × 1011
  --------
      1101
     1101
    0000
   1101
  --------
  10001111
```

结果为 143_{TEN}。

人们在使用二进制数的时候通常将它们写成带有前导零的形式（即第一个 1 的左边有零）。例如 0011，而不是写做 11。这样写不会改变数字的大小，仅仅是为了美观。例如，以下是前 16 个二进制数以及与它们等价的十进制数。

二进制数	十进制数
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

让我们再看一下这一列二进制数。仔细考虑这 4 个垂直列中每一列的 1 和 0，注意它们在一列中自上而下是以怎样的规律交替的。

- 最右边的一列一直在 0 和 1 之间交替。
- 右数第二列是在每两个 0 和两个 1 之间相互交替。
- 下一列是在每四个 0 和每四个 1 之间相互交替。
- 再下一列是在每八个 0 和每八个 1 之间相互交替。

这是很有条理的，难道不是么？事实上，只要再重复这 16 个数字并且在每个数字的前面加一个 1 就可以很容易地写出后面的 16 个数字。

二进制数	十进制数
10000	16
10001	17
10010	18
10011	19
10100	20
10101	21

续表

10110	22
10111	23
11000	24
11001	25
11010	26
11011	27
11100	28
11101	29
11110	30
11111	31

从另外一种角度来看：当以二进制计数的时候，最右边的一位（最低位）以 0 和 1 交替。每当该位由 1 变为 0，从右边数的第二位（次低位）也随之改变——不是由 0 变到 1，就是由 1 变到 0。因此，每次只要有一个二进制数位的值由 1 变到 0，紧挨着的高位数字也会发生变化，而且其变化不是由 0 到 1 就是由 1 到 0。

在书写一个比较大的十进制数的时候，通常在每三位数字之间用一个逗号隔开，这样会让人很清楚地读出数字的大小。例如，当你看到数字 12000000 时，你可能不得不去数一下其中 0 的个数才知道这个数是多少，但是，如果数字写做 12,000,000，你一眼就会知道它是一千二百万。

二进制数的位长度增加得特别快。例如，一千二百万在二进制中应表示为：101101110001101100000000。为了让二进制数更易读，通常在每四个数字之间用一个连字符来分开，例如，1011-0111-0001-1011-0000-0000，或者每四位空出一个空格：1011 0111 0001 1011 0000 0000。本书的后面，我们将看到更简单的二进制数的表示方法。

通过将数字系统减少至只有 0 和 1 两个数字的二进制数字系统，我们已经讨论得足够深入。不可能再找到比二进制数字系统更简单的数字系统了。此外，二进制数字系统还在算术与电子技术之间架起了一座桥梁。在之前的章节中，我们所看到的开关、电线、灯泡，还有继电器等物体，都可以用来表示二进制数 0 和 1。

电线可以表示二进制数字。如果有电流流过这根电线就代表二进制数字 1，如果没有，则代表二进制数字 0。

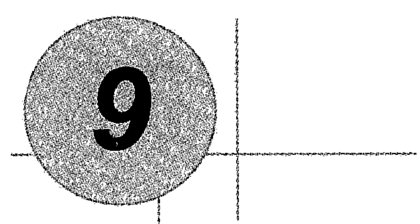
开关可以表示二进制数字。如果开关接通（或闭合）就代表二进制数字 1，如果开关关断（或断开），则代表二进制数字 0。

灯泡可以表示二进制数字。如果灯泡点亮，就代表二进制数字 1，如果没点亮，则代表二进制数字 0。

电报继电器可以表示二进制数字。如果继电器闭合，就代表二进制数字 1，如果断开，则代表二进制数字 0。

二进制数与计算机之间有着紧密的联系。

大约在 1948 年，美国数学家约翰·威尔德·特克（John Wilder Turkey，生于 1915 年）就意识到随着计算机的普及，二进制数很可能会在未来发挥更重要的作用。他决定创造一个新的、更短的词语来代替使用起来很不方便的五音节词——binary digit。他曾经考虑使用 bigit 和 binit，但是最终他还是选用了这个短小、简单、精巧而且非常可爱的词——bit。



二进制数

托尼·奥兰多[1]在他 1973 年所写的一首歌中这样请求他挚爱的人："请在橡树上系上一条黄丝带"。他没有要求爱人进行详细的解释或者进行过多的讨论。他不想听到任何的"如果"、"而且"和"但是"。尽管这首歌是根据那些可能在真实生活中发生过的复杂感情和动人的往事所写的，但这个男人真正想要的答案仅仅是一个简单的"是"或"不是"。他希望在树上系一条黄丝带来表示"是的，尽管你做了很多错事，并且入狱三年，但我依然希望你可以回来和我共同生活。"他希望用树上没有系黄丝带来表示："你连停在这里都别想。"

这是两个界线分明、相互排斥的答案。托尼·奥兰多并没有唱"如果你想考虑一下，

1 托尼·奥兰多（Tony Orlando），美国歌手，1944 年出生于纽约，是一位希腊和波多黎各的混血儿，《老橡树上的黄丝带》是他的经典名曲，脍炙人口。歌曲描述一个在监狱服刑的丈夫在即将服刑期满的前夕，因担心远在家乡的妻子不接受他，写信告诉妻子，如果愿意再次接纳他，请在他出狱返家的当天，在家门口的老橡树上系上一条黄丝带；如果他没有看到黄丝带的话，他就会识相地默默离开。结果这位丈夫出狱返家时，家门口的老橡树上绑满了数以百计的黄丝带在空中飞扬，令他十分感动。后来，绑黄丝带成为全美国的一种风俗，表示迎接久别的亲人回家。——译者注

就系半条黄丝带吧”或者“如果你不再爱我，但我们依然是好朋友，就请系上蓝丝带吧。”相反，他将问题简化得非常非常简单。

还有几个和系黄丝带具有同样效果的例子（但可能无法用在诗里）：选择一种交通标志放在院子外面，来表示“请进”或“此路不通”。

或者在门上挂一个牌子：上面写着“关门”或“营业”。

或者用窗子里的一盏灯的亮灭来表示：“关闭”或“打开”。

如果这就是你想说的，你可以选择很多方法来表示“是”或者“不是”。你不需要用一句话、一个单词甚至一个字母来表达。你所需要的只是一个比特，也就是只需要一个 0 或 1 即可。

就如前面章节所提到的，十进制与其他数字系统相比并没有什么不同，只是我们通常使用它来计数。很明显，我们使用十进制数字系统是因为我们有十个手指。如果我们将数字系统建成八进制数字系统（如果我们是卡通人物）或四进制数字系统（如果我们是龙虾），甚至二进制数字系统（如果我们是海豚），也是合乎情理的。

但是二进制数字系统存在一点特殊性。这个特殊性就在于它是人们所能得到的最简单的数字系统。二进制数字系统中只有两个数字——0 和 1。如果想进一步简化这个数字系统，就只好把 1 去掉，最后我们就剩下的就只有一个数字 0 了。但仅用一个 0 是做不了任何事情的。

单词“bit（比特）”被创造出来代表英文“binary digit”，它的确是和计算机有关的词语中最可爱的一个。当然，这个词也有它一般的意义：“一小部分，程度很低或数量很少”。这个意义很贴切，因为 1 比特——一个二进制数字位——确实是一个非常小的量。

通常当一个新的词语出现的时候，它都会有自己新的意义。比特这个词也是这样的。1 比特的意思并不仅仅是海豚用来计数的二进制数字位所包含的意义。在计算机时代，比特被看做是组成信息块的基本单位。

这是一个大胆的声明。当然，二进制数不是传达信息的唯一方法。字母、单词、莫尔斯码、布莱叶盲文和十进制数都可以来表达信息。关键在于，比特所传递的信息量极少。1 比特是可能存在的最小的信息量。任何小于 1 比特的内容都根本算不上是信息。由

于 1 比特表示的是可能存在的最小信息量，那么复杂一些的信息就可以用多位二进制数来表达（我们说比特传递的信息量“小”，并不是说它传送的信息不重要。事实上黄丝带对于与它相关的两个人来说是一个非常重要的信息）。

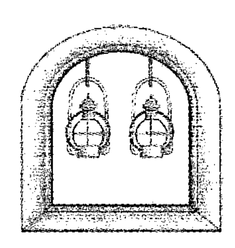
“听，我的孩子，你们很快就能听到保罗・里威尔（Paul Revere）午夜的马蹄声。”亨利・沃兹沃思・朗福罗（Henry Wadsworth Longfellow）写到。尽管他在描述保罗・里威尔是如何通知美国人英国殖民者入侵的消息时不一定与史实完全一致，然而他确实提供了一个耐人寻味的利用二进制传递重要信息的例子：

> 他对他的朋友说：“今晚如果从镇里来的英国军队通过海路或陆路入侵，你就在北教堂钟楼的拱门处高高挂起提灯作为特殊信号——一盏提灯表示从英军从陆路进军，两盏则表示英军从海路入侵……”

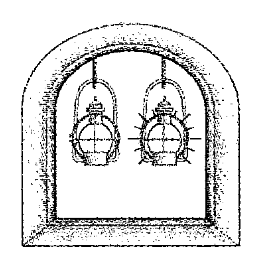
总结起来就是说，保罗・里威尔的朋友有两盏提灯。如果英国军队从陆路进犯，他会在教堂钟楼上挂起一盏提灯。如果英国从海路进军，他会将两盏提灯都在教堂钟楼上挂起。

然而，朗福罗并没有明确地提到所有的可能性。他留下了第三种可能的情况没有说，那就是英军根本没有入侵。朗福罗已经暗示，如果是这样的话可以通过不在教堂钟楼上挂提灯来表示。

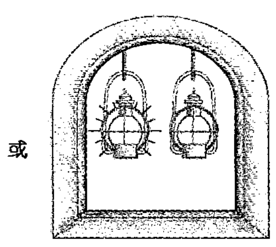
假设两盏提灯实际上是永久固定在教堂钟楼上的。通常它们不会被点亮。



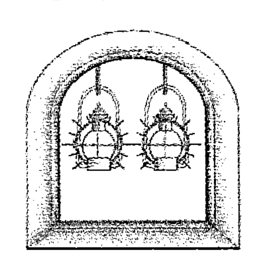
这代表英军还没有入侵。如果一盏提灯亮起：



或



表示英军从陆路入侵。如果两盏灯都被点亮：



表示英军正由海路入侵。

每一盏提灯都代表一个比特。点亮的提灯表示比特值为 1；未点亮的提灯表示比特值为 0。托尼·奥兰多已经向我们证明了传送只有两种可能性的信息只需要一个比特。如果保罗·里威尔只需要被告知英军将要入侵（而不管他们将从何处入侵），一盏提灯就足够了。这盏提灯亮起表示入侵，不亮则表示今夜无事。

要表达包含三种可能性的消息，则还需要再加一盏提灯。如果有了第二盏提灯，这两盏提灯在一起可以表达四种可能的信息：

00 = 英军今晚不会入侵

01 = 英军正由陆路入侵

10 = 英军正由陆路入侵

11 = 英军正由海路入侵

保罗·里威尔将三种可能性用两盏提灯来传送的做法事实上是相当老道的。用通信理论的术语来说，他运用了“冗余”（redundancy）来抵消噪声的影响。通信理论中，噪声（noise）是指影响通信效果的所有事物。电话线上的静电就是一个影响电话通信的鲜明例子。然而，电话通信通常都能成功，因为即使存在噪声，语音中仍存在大量的冗余。我们要理解对方的意思，并不需要听清对方所说的每个词的每个音节。

在上述例子中，噪声是指黑夜中暗淡的光线以及保罗·里威尔距钟楼的距离，这两个因素都能让他分辨不出点亮的是哪盏提灯。以下就是朗福罗诗中至关重要的一段：

哦！他站在与钟楼等高的位置观察，

一丝摇曳的微光，接下来，有一盏灯亮了！

他跳上马鞍，调转马头，

徘徊、凝视，直到看清所有的灯，

钟楼上的第二盏也灯亮了起来！

那当然不是说保罗·里威尔正在分辨到底是哪盏灯先亮的问题。

这里最本质的概念就是，信息是指多个可能性中的一种。例如，当我们与一个人交谈的时候，我们所说的每个字都是对字典中所有字的一个选择。如果我们将所有字典中的字从 1 到 351482 编号，我们可以用数字准确地进行交谈，而无需使用单词。（当然，交谈双方都需要有一本字典，字典上标注着每个数字所代表的是什么字，并且他们还需要有足够的耐心。）

换句话说，所有可以被转换成对两种或多种可能性的选择的信息，都可以用比特来表示。不用说，人类有很多形式的交流是不能用对非此即彼的可能性的选择来表示的，但这些交流形式对我们人类的生存又是至关重要的。这就是人类为什么没有与计算机之间建立起浪漫的关系的原因（无论如何，我们都希望这种情况不会发生）。如果你无法用语言、图画或声音来表达某种事物的时候，你就也无法将这个信息用比特的形式来编码。当然，你也不会想去这么做。

竖起拇指或者不竖起拇指就是一个比特的信息。两个人是否竖起拇指——就如影评人罗杰·艾伯特和刚去世不久的珍妮·西斯科对新影片给出他们的评价时那样——传达了两个比特的信息（我们且不论他们对电影所做了什么评价；我们所关心的只是他们的拇指）。这里有四种可能性，可以用两个比特位来表示：

00 = 他们都不喜欢这部电影

01 = 西斯科讨厌它，艾伯特喜欢它

10 = 西斯科喜欢它，艾伯特讨厌它

11 = 他们都喜欢这部电影

第一个比特位表示西斯科的意见，为 0 表示西斯科讨厌这个部电影，为 1 表示喜欢。同样的，第二位表示艾伯特的意见。

因此，如果你的朋友问你：“对于电影 *Impolite Encounter*，西斯科和艾伯特的评论如何？”你不必回答“西斯科翘起了大拇指，艾伯特却没有。”或者“西斯科喜欢这部电影，而艾伯特不喜欢。”你可以简短地回答：“一零。”只要你的朋友知道哪一位表示西斯科的

意见，哪一位表示艾伯特的意见，并且知道 1 表示竖起拇指，0 表示没有，你的回答就会很容易理解。但是，前提是你和你的朋友都知道编码的含义。

我们也可以一开始就声明值为 1 的比特位表示没有竖起拇指，值为 0 的比特位表示竖起了，这可能有点违反常规。通常我们会认为值为 1 的比特表示赞成，而值为 0 的比特则表示反对，但是实际上这是可以任意调换的。只要每个使用代码的人知道 0 和 1 都具体表示什么意义即可。

某一位或一连串比特位所表示的意义通常是和上下文有关的。系在一棵橡树上的黄色丝带可能仅仅是对于把它系在那里的人和希望在那里看到它的人才有意义。改变丝带的颜色、系丝带的树或者系丝带的日期，它就可能成为一块没有任何意义的破布。同样，想从西斯科和艾伯特的手势中得到有用的信息，我们至少需要知道他们讨论的是哪部电影。

如果你保留了一份西斯科和艾伯特对于电影的评价和投票结果，你就可以在反映他们意见的比特信息中再添加一个比特位，来表示你自己的看法。添加第三个比特位将使得其代表的信息可能性扩展到 8 种：

000 = 西斯科讨厌它，艾伯特讨厌它，我讨厌它
001 = 西斯科讨厌它，艾伯特讨厌它，我喜欢它
010 = 西斯科讨厌它，艾伯特喜欢它，我讨厌它
011 = 西斯科讨厌它，艾伯特喜欢它，我喜欢它
100 = 西斯科喜欢它，艾伯特讨厌它，我讨厌它
101 = 西斯科喜欢它，艾伯特讨厌它，我喜欢它
110 = 西斯科喜欢它，艾伯特喜欢它，我讨厌它
111 = 西斯科喜欢它，艾伯特喜欢它，我喜欢它

利用二进制表示信息的一个额外的好处就是我们可以清楚地知道我们是否已经想到了所有的可能性。我们知道在这种情况下有且仅有 8 种可能性，不多也不少。如果用 3 个比特位，只能从 0 数到 7，后面不会再有其他的三位二进制数存在了。

在描述西斯科和艾伯特意见的比特位中，你可能一直在考虑一个非常重要而且令人不安的问题，这个问题就是：对于李纳德·马丁的“电影及电视指南”我们该怎么办呢？毕竟李纳德·马丁不用手指来评价电影，他用的是一种更传统的星级系统来评价。

要想知道需多少个马丁比特，首先必须了解这个系统的一些情况。马丁给电影的评价是 1～4 颗星，并且中间可以有半颗星，为了好玩，实际上他不会给电影只评一颗星，而是用“BOMB（炸弹）”来代替。这里总共有 7 种可能性，也就意味着我们可以用三个比特位来表示某个评价等级了：

000 = BOMB
001 = ★1/2
010 = ★★
011 = ★★1/2
100 = ★★★
101 = ★★★1/2
110 = ★★★★

你可能会问：“111 呢？”在这里，这个编码不代表任何意义。它没有定义。如果二进制代码 111 被用来表示马丁的评价，那么你就会知道一定是出错了（计算机有可能会出现这样的错误，因为人不会给出这样的评分）。

之前提到过，当我们用两个比特位来表示西斯科和艾伯特的评价时，左边一位表示的是西斯科的意见，右边一位表示的则是艾伯特的意见。在上述马丁的评分系统中，每个单独的比特位都有确定的意义吗？是的，当然有。如果将比特码的数值加 2，再除以 2，我们就得到了马丁评分中对应的星星的数量。但这仅仅是由于我们以一种合乎人们对数字含义体验的方式定义了编码。我们同样也可以将编码作如下定义：

000 = ★★★
001 = ★1/2
010 = ★★1/2
011 = ★★★★
101 = ★★★1/2
110 = ★★
111 = BOMB

只要每个人都明白它的意义，这个编码就与先前的编码同样有效。

如果马丁遇到了一部甚至连一颗星都不值得给的电影，他会给出半颗星。他也当然会有足够的编码来表示半星选项。此时，编码会像下面这样定义：

000 = MAJOR BOMB
001 = BOMB
010 = ★1/2
011 = ★★
100 = ★★1/2
101 = ★★★
110 = ★★★1/2
111 = ★★★★

但是，如果他遇到了一部连半颗星都不值得给的电影，他就决定给它没有星的级别（ATOMIC BOMB？），这时他就得再添加一个比特位了。因为已经没有三个比特位的代码可以用了。

《娱乐周刊》杂志常常举行一些评级活动，评级的对象除了电影之外还有电视节目、CD、书籍、CD-ROM、网站等其他一些东西。等级的范围是从 A+～F（尽管似乎只有波利·舒尔的电影才能堪此殊荣）。计算一下，一共有 13 个可能的等级。我们需要用 4 个比特位来表示这些等级。

0000 = F
0001 = D-
0010 = D
0011 = D+
0100 = C-
0101 = C
0110 = C+
0111 = B-
1000 = B
1001 = B+
1010 = A-
1011 = A
1100 = A+

这里我们有 3 个编码没有用到，即 1101，1110 和 1111，加上这 3 个总共有 16 个编码。

无论何时我们谈到比特，通常所指的都是一定数目的比特位。我们拥有的比特位数

越多，所能表示的不同可能性就越多。

当然，在十进制中也是同样的道理。例如，电话号码的区号有多少位呢？区号共有 3 位十进制数，如果所有的区号都使用的话（实际上我们并没有都使用，但这里不考虑这个问题），一共就会有 10^3，即 1000 个号码，即 000～999。区号是 212 的 7 位电话号码会有多少种可能呢？10^7，即 10,000,000 个。区号是 212，并且以 260 开头的电话号码会有多少个呢？10^4，即 10,000 个。

类似的，在二进制中，可能有的编码数等于 2 的整数次幂，其幂指数就是比特位的位数。

比特位数	编码数量
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
4	$2^4=16$
5	$2^5=32$
6	$2^6=64$
7	$2^7=128$
8	$2^8=256$
9	$2^9=512$
10	$2^{10}=1024$

每增加一个比特位就会将编码的数量增加一倍。

如果已知所要用到编码的数量，如何计算需要多少比特位呢？换句话说，在上表中，如何才能由码字数反推出比特位数呢？

所要用到的方法叫做以 2 为底的对数运算，对数运算是指数运算的逆运算。我们知道 2 的 7 次幂等于 128。那么以 2 为底的 128 的对数就是 7。用数学符号来表示：

$$2^7=128$$

等价于：

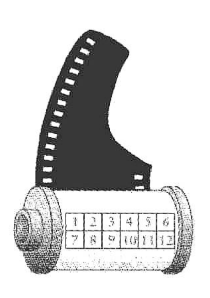
$$\log_2 128=7$$

因此，如果以 2 为底的 128 的对数是 7，以 2 为底 256 的对数就是 8。那么，以 2 为

底 200 的对数是多少呢？大约是 7.64，但实际上并不需要知道具体的值。如果我们要用二进制来表示 200 种不同的事物，共需要 8 个比特位。

比特通常无法从日常观察中找到，它深藏于电子设备中。我们看不到压缩磁盘（CD）、数字手表或计算机中被编码的比特，但有时比特也会清晰地呈现在我们眼前。

下面就是一个例子。如果你有一台相机是使用 35 毫米胶片的，观察一下它的卷轴。这样拿住胶卷：



你可以看到像国际象棋棋盘那样银黑交错排列的方格，这些方格在图中已经用数字 1 ~ 12 进行了标注。这叫做 DX 编码，这 12 个方格实际上是 12 个比特位。一个银色的方格代表值为 1 的比特，一个黑色的方格代表值为 0 的比特。方格 1 和方格 7 通常是银色的（代表 1）。

这些比特代表什么意思呢？你可能会知道，有些胶片比其他胶片对光更敏感些。这种对光的敏感程度称做胶片速度。人们会说某种对光非常敏感的胶片“很快”，那是因为这种胶片的曝光速度极快。胶片的曝光速度是由美国标准协会 ASA（American Standards Association）来制定等级的。最常用的等级有 100、200 和 400。ASA 等级不仅以十进制数字的形式印在胶卷的外包装和暗盒上，它还以比特的形式进行了编码。

胶卷总共有 24 个 ASA 等级，它们是：

25	32	40
50	64	80
100	125	160
200	250	320
400	500	640
800	1000	1250

1600	2000	2500
3200	4000	5000

要对 ASA 等级进行编码需要多少个比特位呢？答案是 5 个。我们知道 2^4 是 16，比 24 要小。2^5 是 32，而这个数又超过了所需的码字数。

与胶片速度对应的比特值如下表所示。

方格 2	方格 3	方格 4	方格 5	方格 6	胶片速度
0	0	0	1	0	25
0	0	0	0	1	32
0	0	0	1	1	40
1	0	0	1	0	50
1	0	0	0	1	64
1	0	0	1	1	80
0	1	0	1	0	100
0	1	0	0	1	125
0	1	0	1	1	160
1	1	0	1	0	200
1	1	0	0	1	250
1	1	0	1	1	320
0	0	1	1	0	400
0	0	1	0	1	500
0	0	1	1	1	640
1	0	1	1	0	800
1	0	1	0	1	1000
1	0	1	1	1	1250
0	1	1	1	0	1600
0	1	1	0	1	2000
0	1	1	1	1	2500
1	1	1	1	0	3200
1	1	1	0	1	4000
1	1	1	1	1	5000

大多数现代的 35 毫米照相机胶片都使用这些码字（除了那些要手动进行曝光的相机和具有内置式测光表但需要手动设定曝光速度的相机以外）。如果你观察一下照相机内部

放胶卷的地方，就会发现在胶卷筒内部有 6 个金属的触点，对应着胶片的金属方格（1～6 号）。银色方格实际上是胶卷暗盒中的金属，即导体；涂有颜料的黑色方格，是绝缘体。

照相机中的电路产生一个电流流入方格 1，方块 1 通常是银色的。这支电流有可能流经（也可能未流经）方格 2～6 上的 5 个触点，这取决于方格是纯银还是涂有颜料的。这样，如果照相机在触点 4 和触点 5 上检测到了电流，而在触点 2、3 和 6 上却没有检测到，那么这个胶片的速度就是 400 ASA。这样，照相机就可以据此来调整胶片的曝光时间了。

假设胶片速度是 50、100、200 或 400 ASA，那么廉价的相机就只需读取方格 2 和方格 3 上的电流。

多数照相机不需要读取或使用方格 8～12。方格 8、9 和 10 被用来对这卷胶卷进行编码，方格 11 和方格 12 指出曝光范围（exposure latitude），这取决于胶片是用于黑白冲洗、彩色冲洗，还是做彩色幻灯片。

也许最常见的二进制数的表现形式就是无所不在的通用产品代码（UPC，Universal Product Code），这个小条形码出现在差不多我们日常所购买的所有商品的包装上。UPC 已经成为计算机逐步进入人们日常生活的标志之一。

尽管 UPC 经常会使人们瞎猜疑，但是它其实是无辜的。使用它的初衷仅仅是为了实现零售业的结算和存货管理的自动化，而且它在这方面的应用也很成功。当 UPC 与一个设计精巧的结账系统结合使用时，顾客会得到一张逐项开列的消费清单，这点是传统的现金出纳员所无法做到的。

有趣的是，UPC 也是二进制码，尽管乍看起来它并不太像。将 UPC 解码并看看 UPC 到底是怎样工作的，就明白了。

在最常见的形式中，UPC 是由 30 条不同宽度的垂直黑色条纹组成的，它们的间隔宽度也不同，条纹下面标有数字。例如，以下是 Campbell 公司出品的 10 3/4 盎司的罐装鸡汁面汤包上的 UPC。



我们要试着将条形码形象地看成是细条和黑条、窄间隙和宽间隙的排列，事实上，这就是观察条形码的一种方式。在 UPC 中，黑色条纹有四种不同的宽度，宽条纹的宽度分别是最细条纹宽度的两倍、三倍或四倍。同样，宽间隔的宽度分别是最窄间隔宽度的两倍、三倍或四倍。

但是，另一种解读 UPC 的方式就是将它们看做一列比特位。记住整个条形码不是在收银台的扫描仪所“看到”的那样。扫描仪所读到的并不是下面一排的数字，因为这需要更精密的光学识别（OCR，Optical Character Recognition）技术。扫描仪只识别整个条形码的一条窄带，条形码做得很大是为了便于结算台的操作人员用扫描仪对准。扫描仪所看到的那一个 UPC 断面可以这样表示：

看起来与莫尔斯码很像，不是吗？

当计算机从左向右扫描这个信息时，它会首先给遇到的第一个黑条分配一个值为 1 的比特，给与这个黑条相邻的白色间隙分配一个值为 0 的比特。随后的条纹和间隙被读做一行中的一系列比特，每个系列的比特可以是 1 位、2 位、3 位或 4 位，而这个位数取决于条纹和空隙的宽度。本例中扫描仪所扫描到的条形码与比特位之间的关系可以简单的表示为：

10100011010110001001100100011010001101000110101010111001011001101101100100111011001101000100101

因此，整个 UPC 只不过是一串 95 位二进制数。在本例中，这些比特可以做如下分组。

前 3 位通常都是 101，这就是最左边的护线，它帮助计算机扫描仪定位。从护线中，扫描仪可以确定代表单个比特的条和间隙的宽度是多少。否则，所有包装上的 UPC 就都得采用指定的大小了。

最左边的护线之后是 6 组比特串，每串含有 7 个比特位。其中每一组都可以是数字 0～9 的编码，后面我会做一个简短的说明。接下来是一个 5 比特位的中间护线，这是一个固定的模式（始终是 01010），它是一个内置式的检错码。如果计算机扫描仪没有在应有的位置找到中间护线，它就无法破解 UPC 码。这条中间护线是用来预防条形码被篡改或被印错的一种方法。

比特	意义
101	最左边的护线
0001101 0110001 0011001 0001101 0001101 0001101	左边的数字
01010	中间的护线
1110010 1100110 1101100 1001110 1100110 1000100	右边的数字
101	最右边的护线

中间护线后面仍然是 6 组比特串，每组中含有 7 个比特位。之后是最右边的护线，最右边的护线通常都为 101。最右边的护线可以实现 UPC 的反向扫描（也就是从右到左扫描），这一点我们将在后面解释。

因此，整个 UPC 对 12 个数字进行了编码。UPC 的左边含有 6 个编码数字，每个数字占有 7 个比特位。你可以利用如下的表格来解码。

左边的编码

0001101 = 0	0110001 = 5
0011001 = 1	0101111 = 6
0010011 = 2	0111011 = 7
0111101 = 3	0110111 = 8
0100011 = 4	0001011 = 9

注意，这里每个 7 位编码都是以 0 开头，以 1 结尾的。如果扫描仪遇到了一个位于左边的 7 位编码，这个编码是以 1 开头以 0 结尾的，那么它就知道自己没有将 UPC 正确地读入或者条形码被篡改了。另外我们还注意到每组编码都仅有两组连续为 1 的比特位，这就暗示每个数字对应着 UPC 码中的两个垂直条纹。

你也会发现，上表中每组编码都含有奇数个 1。这是另一种检查错误和一致性的方法，

称为奇偶校验。如果一组比特位中含有偶数个 1，它就称为偶校验；如果含有奇数个 1，那么它就称为奇校验。这样看来，所有这些编码都拥有奇校验。

破解右边的 6 组 7 位编码要用到下表。

右边的编码

1110010 = 0	1001110 = 5
1100110 = 1	1010000 = 6
1101100 = 2	1000100 = 7
1000010 = 3	1001000 = 8
1011100 = 4	1110100 = 9

这些编码都是之前编码的补码：之前出现 0 的地方，现在都换成了 1，反之亦然。这些编码都是以 1 开头，以 0 结尾的。除此之外，每组编码都含有偶数个 1，属于偶校验。

现在，我们就可以解读 UPC 了。运用以上两个表格，我们可以确定，Campbell 公司的 10 3/4 盎司罐装鸡汁面汤包上的 12 个数字为：

0　51000　01251　7

太令人失望了。就如你所看到的，这与 UPC 下面印刷的数字完全相同（这样做是很有意义的，在扫描仪由于某种原因无法解读出条形码时，收银员就可以手动输入这些数字，毫无疑问你也看到过这一幕）。我们没有必要了解解码的全部过程，况且，我们也无法从中解码出任何秘密信息。不过，关于 UPC 的解码工作已经没有什么可做了，那 30 根竖条已经变成了 12 个数字。

第一个数字（在这里是 0）被称为数字系统符。0 意味着这是一个常规的 UPC。如果 UPC 出现在杂货店中那些需要称重的商品上，例如肉、农产品，这个编码就会是 2。票券的 UPC 的第一个数字通常是 5。

接下来的 5 个数字表示制造商编码。这种情况下，51000 就是 Campbell 公司的编码。所有 Campbell 公司的产品都有这个编码。后面的 5 位代码（01251）是这个公司的某种商品的编码，上例中这个数字就是指 10 3/4 盎司罐装鸡汁面。只有和制造商编码同时出现的时候这个编码才有意义。另一个公司的鸡汁面会有另外一个编码，01251 在另外一个公

司可能是指一种完全不同的产品。

与大家通常的想法相反，UPC 不包含物品的价格信息。价格信息可以从商店使用的与该扫描仪相连的计算机中检索到。

最后一个数字（在这里是 7）称为模校验字符。这个字符可用来进行另外一种错误检验。为了了解它是如何工作的，我们将前 11 个数字（在这个例子中是 0 51000 01251）各用一个字母来代替：

A　BCDEF　GHIJK

然后，计算下式的值：

$$3\times(A+C+E+G+I+K)+(B+D+F+H+J)$$

从离这个值最近并且大于或等于它的一个 10 的整倍数中减去它，其结果称为模校验字符（modulo check character）。在 Campbell 鸡汁面的例子中，有：

$$3\times(0+1+0+0+2+1)+(5+0+0+1+5)=3\times4+11=23$$

紧挨 23 并大于或等于 23 的 10 的整倍数是 30，因此：

$$30-23=7$$

这就是印在外包装上并以 UPC 形式编码的模校验字符，这是一种冗余措施。如果扫描仪没有扫描到与计算机计算结果相同的模校验字符，那么计算机就视这个 UPC 无效。

通常情况下，要表示 0～9 的十进制数字只需要 4 个比特位就足够了。UPC 中每个数字用了 7 个比特位。这样，UPC 总共用了 95 个比特位来表示 11 个有效的十进制数。实际上，UPC 中还有空白位置（相当于 9 个 0 比特），它们位于左、右护线的两侧。这就意味着，整个 UPC 需要 113 个比特位来编码 11 个十进制数，平均每个十进制数所用的比特位超过了 10 个！

如我们所看到的，有部分冗余对于检错来讲是必要的。这种商品编码如果能够被顾客用笔轻易地改动，那么这种产品编码措施也就没有任何意义了。

UPC 可以从两个方向读，这一点是非常方便的。如果扫描装置解码的第一个数是符合偶校验（即 7 位编码中有偶数个 1）的，扫描仪就会知道，它正从右向左扫描 UPC 码。

计算机就会使用如下表来解码。

逆向扫描时右边数字的编码

0100111 = 0	0111001 = 5
0110011 = 1	0000101 = 6
0011011 = 2	0010001 = 7
0100001 = 3	0001001 = 8
0011101 = 4	0010111 = 9

以下是对左边数字的解码表。

逆向扫描时左边数字的编码

1011000 = 0	1000110 = 5
1001100 = 1	1111010 = 6
1100100 = 2	1101110 = 7
1011110 = 3	1110110 = 8
1100010 = 4	1101000 = 9

所有这些 7 位编码都与由左向右扫描时得到的 UPC 完全不同。这里不会有模棱两可的现象存在。

在这本书中我们是从莫尔斯码开始接触编码的，莫尔斯码是由点和划组成的，且点和划之间由一定的时间间隔分开。尽管莫尔斯码乍看上去并不像是由 0 和 1 组成的，然而它们却是等价的。

回顾一下莫尔斯码的编码规则：划的长度等于点长度的三倍；单个字母内，点或划之间以长度与点相等的空格来分开；单词内的各个字母之间用长度等于划的空格分隔；各单词之间由长度等于两倍的划的空格分开。

让我们将分析简化一些，假设一个划的长度是点长度的两倍，而不是三倍。这就意味着一个点就是一个值为 1 的比特位，一个划是两个值为 1 的比特位，空格则是值为 0 的比特位。

下面是第 2 章中莫尔斯码的基本表。

A	•—	J	•———	S	•••
B	—•••	K	—•—	T	—
C	—•—•	L	•—••	U	••—
D	—••	M	——	V	•••—
E	•	N	—•	W	•——
F	••—•	O	———	X	—••—
G	——•	P	•——•	Y	—•——
H	••••	Q	——•—	Z	——••
I	••	R	•—•		

以下是转化为比特的结果。

A	101100	J	1011011011100	S	1010100
B	1101010100	K	110101100	T	1100
C	11010110100	L	1011010100	U	10101100
D	11010100	M	1101100	V	1010101100
E	100	N	110100	W	101101100
F	1010110100	O	11011011100	X	11010101100
G	110110100	P	10110110100	Y	110101101100
H	101010100	Q	110110101100	Z	11011010100
I	10100	R	10110100		

注意，这里所有编码都是以 1 开头，以两个 0 结尾。结尾处的两个 0 代表单词内各个字母之间的空格。单词之间的空格编码则是另外一对 0。因此，在莫尔斯码中 "hi there" 通常是这样表示的：

但是，当莫尔斯码采用比特来表示的时候，看起来就像 UPC 的一个截面。

101010100101000011001010101001001011010010000

用比特的形式表示布莱叶盲文比表示莫尔斯码容易得多。布莱叶盲文是 6 位编码。其中的每一个字母都是由 6 个点组成的，这些点可能是凸起的，也可能是不凸起的。就如同在第 3 章中所提到的那样，这些点通常用数字 1～6 编号。

1 ○ ○ 4
2 ○ ○ 5
3 ○ ○ 6

例如，单词“code”用布莱叶盲文可以这样表示：

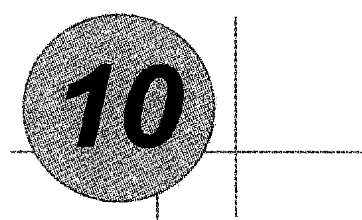
如果突起的点代表 1，平坦的点代表 0，则在布莱叶盲文中，每个符号都可以用 6 个比特的二进制代码来表示。单词“code”中的四个布莱叶字母符号可以简单地写成：

100100 101010 100110 100010

这里，最左边的一位对应编号为 1 的位置，最右边的一位对应编号为 6 的位置。

在本书后面的章节可以看到，比特可以表示文字、图片、声音、音乐、电影，也可以表示产品编码、胶片速度、影评结果、英国军队的入侵，以及心爱之人的意图。但是，从根本来说，比特是数字。在用比特表示其他信息的时候我们所要做的就是计算有多少种可能性。这决定了我们需要的比特位数，以便每种可能性都可以分配到一个编号。

比特在逻辑学中也很重要。逻辑学是哲学和数学的奇特融合，其主要目的就是确定某个陈述是真还是假。真和假同样可以表示为 1 和 0。



逻辑与开关

什么是真理？亚里士多德认为真理是与逻辑相关的某种事物。《工具论》（可以追溯到公元前 4 世纪）这本书收集了他的理论，这也是最早在逻辑上做出全面阐述的著作。对于古希腊人来说，逻辑是在追求真理的过程中的一种分析方法，因此它被人们认为是一种哲学形式。亚里士多德的逻辑学基础是三段论法。最著名的三段论法（但是这个三段论却并没有出现在亚里士多德的著作中）是：

所有男人都必有一死；

苏格拉底是男人；

因此，苏格拉底必有一死。

在三段论中，首先假定两个条件是正确的，然后再通过这两个条件推断出结论。

苏格拉底之死这种结论似乎太简单了，然而，三段论的形式其实是多种多样的。例如，以下两个由 19 世纪的数学家查尔斯·道奇森（也被人们称为刘易斯·卡罗尔）提出的条件：

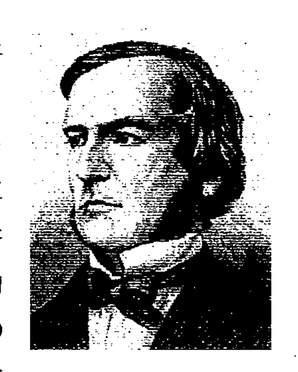
所有哲人都是符合逻辑的；

一个没有逻辑的人总是非常顽固的。

上述条件的推论一点也不明显（结论是“一些顽固的人不是哲人。”注意，这里“一些”这个词的出现完全是意料之外的）。

在过去的两千多年里，数学家们都压制着亚里士多德的逻辑学，并试图用数学符号和算子限制它的发展。19 世纪前期，唯一涉足这个领域的人就是莱布尼茨（1648-1716），他早年涉足了逻辑学，而后转移到了其他兴趣上（例如，与牛顿在同一时期，独立发明微积分）。

而后乔治·布尔出现了。



乔治·布尔于 1815 年出生于英格兰，降临到了这个对于他来说充满不公的世界。乔治的父亲是个鞋匠，母亲曾经做过女佣，因此，按常理在当时英国森严的等级制度下，他基本上不会做出什么有别于父辈们的事业。但是他凭借着勇于探究的精神以及父亲的帮助（乔治的父亲对科学、数学和文学都有浓厚的兴趣），乔治在年少的时候就接受了良好的教育，而这本应是上流社会中的男孩子们才能享有的特权。他学习了拉丁语、希腊语和数学。1849 年，布尔凭借着他早期在数学研究上的论文成为了爱尔兰利克市皇后书院的数学系首席教授。

19 世纪中叶的一些数学家一直在研究逻辑的数学定义（其中最著名的是奥古斯特·德·摩根），但在理论上有着实际突破的却是布尔。首先是一部短篇著作《逻辑的数学分析——关于演绎推理的一篇随笔》（*The Mathematical Analysis of Logic, Being an Essay Towards a Calculus of Deductive Reasoning*，1847 年），而后是一篇更长且更宏大的著作《思维规律的研究——逻辑与概率数学理论的基础》（*An Investigation of the Laws of Thought on Which Are Founded the Mathematical Theories of Logic and Probabilities*，1854 年），通称《思维规律》（*The Laws of Thought*）。布尔死于 1864 年，他由于冒雨赶去上课而感染了肺炎，那一年，他年仅 49 岁。

布尔于 1854 年所发表著作，其书名就彰显了一种雄心壮志。因为理性的人总是通过逻辑去进行思考，如果我们找到了一种利用数学来描述逻辑的方法，也会找到一种用数学方法来描述大脑是如何工作的。当然，现在这种观点显得很是天真（然而在那个时代这确实是非常先进的思想）。

布尔发明了一种代数，这种代数看上去与传统代数非常相似，而且运算规则也非常类似。在传统的代数中，操作数（通常为字母）代表数字，算子（通常为“+”和“×”）则用来指示这些数字之间如何运算。通常，我们应用传统代数来解决可能像这样的问题：安雅有 3 磅豆腐。贝蒂的豆腐是安雅的 2 倍。卡门的豆腐比贝蒂多 5 磅。迪尔得丽的豆腐是卡门的 3 倍。试问迪尔得丽有多少豆腐？

在解决这个问题时，我们首先要将文字叙述转化为数学语言，下面用四个字母分别表示每个人所拥有的豆腐有多少磅：

$$A = 3$$

$$B = 2 \times A$$

$$C = B + 5$$

$$D = 3 \times C$$

将以上 4 个表达式带入同一式子中合并，最终可以得到一个加法和乘法混合运算的式子：

$$D = 3 \times C$$

$$D = 3 \times (B + 5)$$

$$D = 3 \times ((2 \times A) + 5)$$

$$D = 3 \times ((2 \times 3) + 5)$$

$$D = 33$$

当进行传统代数运算的时候，我们会遵循一定的规则。这些规则在实际中可能非常根深蒂固，以至于我们不再认为它们是规则，甚至可能忘记它们的名字。但规则确实是任何形式数学运算的基础。

首先是加法和乘法的交换律（commutative）。意思就是，在运算符两边的操作数可以任意调换：

$$A + B = B + A$$

$$A \times B = B \times A$$

相反，减法和除法则是无法应用交换律的。

加法和乘法也遵循结合律（associative），如下：

$$A + (B + C) = (A + B) + C$$
$$A \times (B \times C) = (A \times B) \times C$$

最后，乘法遵循加法分配率（distributive）：

$$A \times (B + C) = (A \times B) + (A \times C)$$

传统代数的另一个特点就是，它是处理数字的，例如，豆腐的重量、鸭子的数量、火车行驶的距离或家庭成员的年龄。布尔的天才之处就在于他把代数从数的概念中抽离出来而使其更加抽象。在布尔代数（Boole's algebra，现在也这样叫）中，操作数不是数字而是类（class）。简单说，一个类就是一个事物的群体，它后来也被称为集合（set）。

以猫为例。猫可以分为公猫与母猫。为了简便，用字母 ***M*** 代表公猫，字母 ***F*** 代表母猫。记住，这两个符号代表的不是数字。公猫与母猫的数量会随着小猫的出生和老猫的死去而发生变化。这两个字母分别表示两类猫——有特定特征的猫的群体。当我们提到公猫的时候，就可以使用 ***M***。

我们也可以用别的字母代表猫的颜色，例如：***T*** 可以代表褐色的猫，***B*** 可以代表黑猫，***W*** 代表白猫，***O*** 代表不在 ***T***、***B*** 或 ***W*** 集合中的其他颜色的猫。

仍以此为例，猫还可以分为已被绝育的和未被绝育的。我们用字母 ***N*** 来表示已被绝育的猫，字母 ***U*** 表示未被绝育的猫。

在传统的（数字的）代数中，符号“+”和“×”用来表示加法和乘法。在布尔代数中，也有“+”和“×”这样的符号，这很可能造成混淆。所有人都知道在传统代数中如何将数字相加或相乘，但是我们如何将类相加或相乘呢？

然而，在布尔代数中，并没有实际意义上的加或乘。而符号“+”和“×”表示的是完全不同的意义。

在布尔代数中，符号“+”表示两个集合的并集。两个集合的并集的意思就是指第一个集合中的所有元素与第二个类中所有元素的集合。例如，***B+W*** 表示的就是所有黑猫和白猫的集合。

在布尔代数中，符号“×”表示两个集合的交集。两个集合的交集就是指既在第一个集合中又在第二个集合中的所有元素的集合。例如，$\boldsymbol{F} \times \boldsymbol{T}$ 代表的是所有褐色母猫的集合。像传统代数一样，我们可以将 $\boldsymbol{F} \times \boldsymbol{T}$ 写为 $\boldsymbol{F \cdot T}$，或者简写为 $\boldsymbol{FT}$（布尔代数中的首选形式）。可以将两个字母想象成两个形容词串联在一起：褐色的母猫。

为了避免在传统代数与布尔代数间混淆，有时用符号“∪”和“∩”来代替“+”和“×”。布尔对数学方面的革命性影响是让人们熟知的符号更加抽象，因此，我决定坚持他的做法，而不是引入新的符号到他的代数中。

交换律、结合律和分配律都在布尔代数中同样成立。而且在布尔代数中，加法还可以来分配乘法，但在传统的代数中，这是不成立的：

$$\boldsymbol{W} + (\boldsymbol{B} \times \boldsymbol{F}) = (\boldsymbol{W} + \boldsymbol{B}) \times (\boldsymbol{W} + \boldsymbol{F})$$

白猫和黑色母猫的并集与如下两个集合的交集是一样的，这两个集合分别是，白猫和黑猫的并集，白猫和母猫的并集。虽然有点难理解，但是，这个结论是成立的。

布尔代数中还有另外两个符号是非常重要的。这两个符号看起来像数字，但是它们并不是真正意义上的数字，相对于数字而言，它们有不同的意义。在布尔代数中，符号 1 表示“全集”——也就是我们所提到的所有事物。在前面的例子中，符号 1 表示的就是“所有猫的集合”。因此：

$$\boldsymbol{M} + \boldsymbol{F} = 1$$

也就是说，公猫与母猫的并集是所有猫。同样的，褐色的猫、黑猫、白猫与其他颜色猫的并集也是所有猫的集合：

$$\boldsymbol{T} + \boldsymbol{B} + \boldsymbol{W} + \boldsymbol{O} = 1$$

通过以下方法也可以得到所有猫的集合：

$$\boldsymbol{N} + \boldsymbol{U} = 1$$

符号 1 与减号连用可以表示在全集中排除一些事物，例如，

$$1 - \boldsymbol{M}$$

意思就是除去公猫的所有猫的集合。这个集合与母猫的集合是相等的：

$$1-\boldsymbol{M}=\boldsymbol{F}$$

另一个要用到的符号就是 0，在布尔代数中，符号 0 表示空集——不包含任何元素的集合。空集往往是两个互斥集合的交集，例如，母猫与公猫的交集：

$$\boldsymbol{F}\times\boldsymbol{M}=0$$

注意，在布尔代数中，1 和 0 有时也同在传统代数中的应用一样。例如，所有猫与母猫的交集是母猫的集合：

$$1\times\boldsymbol{F}=\boldsymbol{F}$$

空集与母猫的交集还是空集：

$$0\times\boldsymbol{F}=0$$

空集与母猫的并集则是母猫：

$$0+\boldsymbol{F}=\boldsymbol{F}$$

但是有时也会出现与传统代数相悖的结果。例如，所有猫与母猫的并集是所有猫的集合：

$$1+\boldsymbol{F}=1$$

这个式子在传统代数中就是没有意义的。

由于 $\boldsymbol{F}$ 是母猫的集合，($1-\boldsymbol{F}$) 是所有非母猫的猫的集合，因此这两个集合的并集是 1：

$$\boldsymbol{F}+(1-\boldsymbol{F})=1$$

而这两个集合的交集为 0：

$$\boldsymbol{F}\times(1-\boldsymbol{F})=0$$

在历史上，这个公式代表了逻辑学上的一个重要概念：这一概念被称为矛盾律。矛盾律指出事物不可能既是它本身，同时又是它的对立面。

布尔代数中与传统代数形式上最大区别之处就是这样一个表达式：

$$F \times F = F$$

这个表达式很明确地表达了布尔代数的意义：母猫和母猫的交集依然是母猫。但是如果 F 代表的是数字，这个表达式就不会成立。布尔认为式子：

$$X^2 = X$$

就是将其代数同传统代数区别开的一条语句。另一个在传统代数中看起来比较有趣的式子是：

$$F + F = F$$

母猫和母猫的并集依然是母猫。

布尔代数提供了一种解决亚里士多德三段论的数学方法。我们再看看最著名的三段论法中的前两句，但是现在不区分性别：

所有人都难逃一死；

苏格拉底是人。

我们用 P 表示所有人的集合，M 表示必有一死的事物，S 表示苏格拉底的集合。

“所有人都难逃一死”是什么意思？意思就是说所有人的集合与必有一死的事物的交集是所有人：

$$P \times M = P$$

命题 $P \times M = M$ 是错误的，因为，在必有一死的事物中，还包含猫、狗和榆树等。

“苏格拉底是人”这句话的意思就是苏格拉底的集合（一个很小的集合）与所有人的集合（一个很大的集合）的交集是苏格拉底：

$$S \times P = S$$

根据第一个等式我们可以得到 $P = P \times M$，我们将这个等式带入到第二个等式中去：

$$S \times (P \times M) = S$$

根据结合律，可以写为：

$$(\boldsymbol{S} \times \boldsymbol{P}) \times \boldsymbol{M} = \boldsymbol{S}$$

并且我们已知 $\boldsymbol{S} \times \boldsymbol{P} = \boldsymbol{S}$，因此上式可以简化为：

$$\boldsymbol{S} \times \boldsymbol{M} = \boldsymbol{S}$$

到此，我们得出了结论。这个表达式告诉我们，苏格拉底的集合与必有一死的事物的集合的交集是 $\boldsymbol{S}$，也就是说，苏格拉底必有一死。如果我们让 $\boldsymbol{S} \times \boldsymbol{M} = 0$，会得出苏格拉底不会死。如果 $\boldsymbol{S} \times \boldsymbol{M} = \boldsymbol{M}$，结论就会是：只有苏格拉底会死，其他任何事物都是不死的！

应用布尔代数证明这个显而易见的事实（实际上在2400年前苏格拉底自己就证明了他必有一死）似乎有点小题大做，然而布尔代数同样可以用来确定某种事物是否遵循特定的标准。或许某天你走进了一家宠物商店，对店员说："我想要一只公猫，已绝育的，白色或褐色都可以，或者一只母猫，也要是已绝育的，除了白色任何颜色都可以；或者，只要是黑猫就可以。"店员会对你说："你想要的猫是在以下这样的集合里：

$$(\boldsymbol{M} \times \boldsymbol{N} \times (\boldsymbol{W} + \boldsymbol{T})) + (\boldsymbol{F} \times \boldsymbol{N} \times (1 - \boldsymbol{W})) + \boldsymbol{B}$$

对么？"你会说："是的！正是！"

为了确定店员是正确的，你大概会去抛弃并集和交集的概念，并用 OR 和 AND 取而代之。这里将字母大写是因为，它们表示的不仅是字面意义，而表示布尔代数中的运算。做并集的时候，可以看做：第一个集合 OR 第二个集合。做交集的时候，可以看做：第一个集合 AND 第二个集合。除此之外，NOT 可以看做在 1 后面加一个减号。总的来说：

- 符号"+"（之前作为并集的符号）现在可以用 OR 来表示。
- 符号"×"（之前作为交集的符号）现在可以用 AND 来表示。
- 符号"1−"（之前意思是从全集中去掉某些元素）现在用 NOT 来表示。

因此，原表达式可以写为：

$$(\boldsymbol{M} \text{ AND } \boldsymbol{N} \text{ AND } (\boldsymbol{W} \text{ OR } \boldsymbol{T})) \text{ OR } (\boldsymbol{F} \text{ AND } \boldsymbol{N} \text{ AND } (\text{NOT } \boldsymbol{W})) \text{ OR } \boldsymbol{B}$$

这样就非常接近你所说的话了。注意，这里是如何用括号表述清楚你的意图的。你想要的猫来自以下三个集合中的一个：

$$(\boldsymbol{M} \text{ AND } \boldsymbol{N} \text{ AND } (\boldsymbol{W} \text{ OR } \boldsymbol{T}))$$
$$\text{OR}$$
$$(\boldsymbol{F} \text{ AND } \boldsymbol{N} \text{ AND } (\text{NOT } \boldsymbol{W}))$$
$$\text{OR}$$
$$\boldsymbol{B}$$

有了这个公式，店员就可以做一个布尔测试了。为了避免麻烦，我采用了一种略微有点不同的布尔代数形式——字母不仅仅代表集合。这里，字母可以用数字来赋值。我们只用数字 0 和 1。数字 1 代表 YES，True，即这只猫是符合这样的标准的。数字 0 表示 NO，False，即这只猫不符合这种特定标准。

首先，店员拿出了一只未绝育的褐色公猫。以下是我期望得到的猫的表达式：

$$(\boldsymbol{M} \times \boldsymbol{N} \times (\boldsymbol{W} + \boldsymbol{T})) + (\boldsymbol{F} \times \boldsymbol{N} \times (1 - \boldsymbol{W})) + \boldsymbol{B}$$

以下是用 0 和 1 替换之后的式子：

$$(1 \times 0 \times (0 + 1)) + (0 \times 0 \times (1 - 0)) + 0$$

注意，只有 ***M*** 和 ***T*** 的值为 1，因为这只猫是褐色的公猫。

现在我们要做的就是化简这个表达式。如果简化结果为 1，则这只猫就符合你的标准；如果简化结果为 0，那么这只猫就不符合。在化简的时候要切记，我们并不是真的在进行加和乘的运算，尽管通常我们可以当做是。符号“+”代表 OR，符号“×”代表 AND，在大多同类的规则中适用（有时在现代课本中也使用符号“∧”和“∨”来表示 AND 和 OR，而非用符号“×”和“+”。但是在这里符号“+”和“×”是具有特殊意义的）。

当符号“×”代表 AND 时，有以下几种可能的结果：

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

也就是说，只有当运算符 AND 左右两端都为 1 的时候，结果才为 1。这个运算与常规的乘法是完全相同的，该规则可以总结为如下这样一个表格，这与第 8 章中加法和乘

法表的形式类似。

AND	0	1
0	0	0
1	0	1

当符号“+”表示 OR 的时候，有以下几种可能的结果：

$$0+0=0$$

$$0+1=1$$

$$1+0=0$$

$$1+1=1$$

如果运算符 OR 左右两端有一个操作数为 1，则其运算结果就为 1。这个运算与常规的加法类似，但是在 1 + 1 = 1 这里例外。运算符 OR 可以概括为如下这样一个表格。

OR	0	1
0	0	1
1	1	1

我们应用这两个表格来计算一下原表达式的结果：

$$(1\times 0\times 1)+(0\times 0\times 1)+0=0+0+0=0$$

结果为 0，意味着 NO，False，这只小猫不符合标准。

接着，店员拿出了一只已绝育的白色母猫。原始表达式是：

$$(\boldsymbol{M}\times\boldsymbol{N}\times(\boldsymbol{W}+\boldsymbol{T}))+(\boldsymbol{F}\times\boldsymbol{N}\times(1-\boldsymbol{W}))+\boldsymbol{B}$$

将 0 和 1 带入得：

$$(0\times 1\times(1+0))+(1\times 1\times(1-1))+0$$

化简得：

$$(0\times 1\times 1)+(1\times 1\times 0)+0=0+0+0=0$$

因此，第二只小猫也不符合标准。

然后，店员拿出了一只已绝育的灰色母猫（灰色属于其他颜色——非白色、黑色或褐

色）。表达式如下：

$$(0 \times 1 \times (0 + 0)) + (1 \times 1 \times (1 - 0)) + 0$$

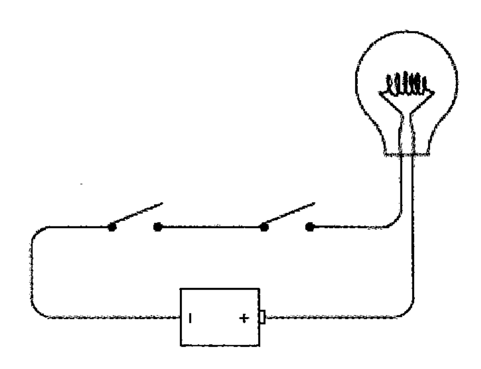
化简结果为：

$$(0 \times 1 \times 0) + (1 \times 1 \times 1) + 0 = 0 + 1 + 0 = 1$$

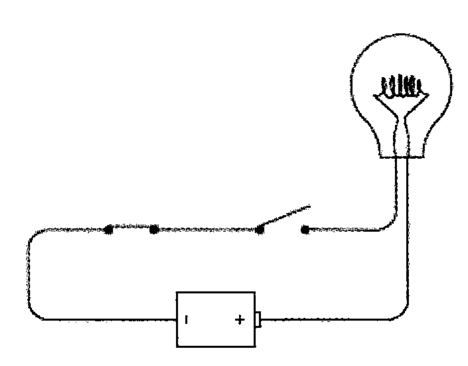
最后结果为 1，意味着 Yes，True，这个只小猫可以带回家（而且还是只漂亮的小猫！）。

当晚，在小猫蜷缩在你的腿上睡觉的时候，你突发奇想是否可以通过连通开关和灯泡的方法来确定某类猫咪是否符合你的标准（是的，你就是个奇怪的孩子）。你根本没发现，你将做出一个重要的概念上的突破。你要做的这些实验将布尔代数与电路相融合，并且可以设计和制造利用二进制进行计算的计算机。但是，不要让这些吓倒你。

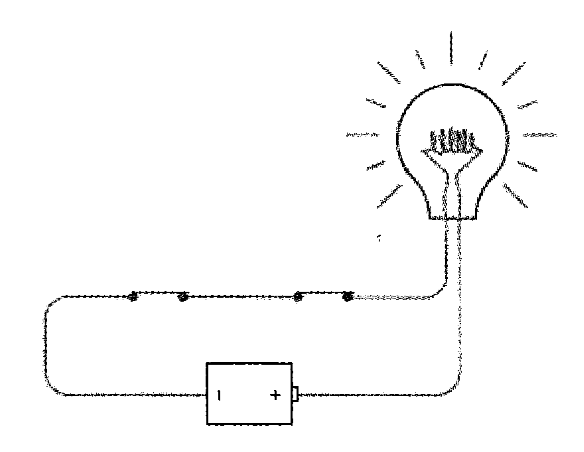
在实验的开始，将灯泡和电池正常连接起来，但是，你用了两个开关，而不是一个，如下图所示。



像这样一个接着一个首尾相连的接线方式称做串联（series）。如果你闭合左端的开关，什么都不会发生。



如果让左边的开关保持断开，而闭合右边的开关，同样什么都不会发生。只有当左右两个开关都闭合时灯泡才会亮，如下图所示。



这里的关键词是“与”。当左右两个开关必须都是闭合的时候电流才流过回路。

这个电路是一个简单的逻辑演示。实际上，灯泡是在回答“两个开关是否都闭合？”这样一个问题。这个电路的运转情况，我们可以总结为如下表格。

左开关	右开关	灯泡
断开	断开	不亮
断开	闭合	不亮
闭合	断开	不亮
闭合	闭合	亮

在之前的章节中，我们已经知道二进制数是如何表示信息的——而信息是无所不包的，无论是简单的数字还是罗杰·艾伯特[2]拇指的方向都是信息。我们可以说二进制 0 代表“艾伯特的拇指向下”，二进制 1 代表“艾伯特的拇指向上。”一个开关有两个状态，因此可以代表二进制数。我们可以说 0 代表“开关断开”，1 代表“开关闭合”。一个灯泡有两种状态，因此，它也可以用二进制数来表示。我们可以说 0 代表“灯泡不亮”，1 代表“灯泡亮”。根据以上表述，我们可以将表格简化为如下形式。

2 美国影评家—译者注

左开关	右开关	灯泡
0	0	0
0	1	0
1	0	0
1	1	1

注意，如果将左边的开关和右边的开关调换位置，结果是一样的。我们不用分辨开关哪个是哪个。所以，上面的表也可以写为和 AND 表或 OR 表类似的形式。

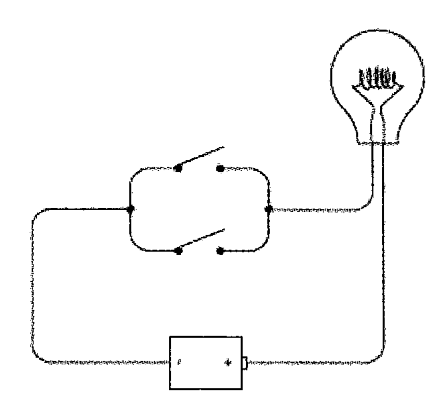
串联的开关	**0**	**1**
0	0	0
1	0	1

的确，这与 AND 表是一样的。

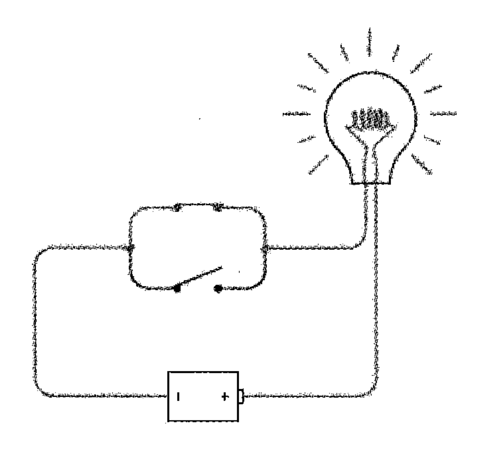
AND	**0**	**1**
0	0	0
1	0	1

这个简单的电路演示了布尔代数中的 AND 运算。

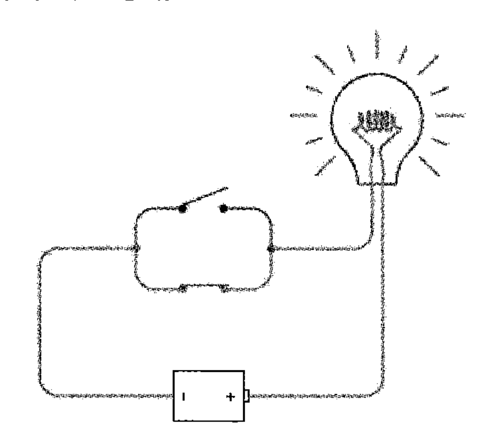
接下来，稍微改变一下开关连接方式，如下图所示。



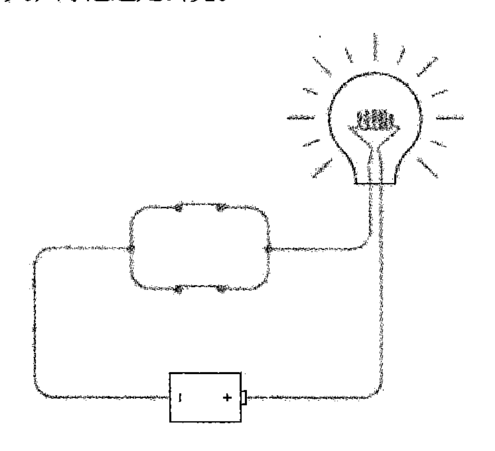
这种连接方式叫做并联（parallel）。这种连接与上一种连接的不同之处就在于，闭合上面的开关，灯泡就会亮。



或者闭合下面的开关，灯泡也会亮。



或者闭合所有的开关，灯泡还是会亮。



如果上面的开关闭合或者下面的开关闭合或者都闭合，灯泡都会亮。这里的关键词为“或”。

电路又一次做了一个逻辑上的演示。灯泡回答了“是否有开关闭合？”的问题。下表总结了这样一个电路的工作原理。

上开关	下开关	灯泡
断开	断开	不亮
断开	闭合	亮
闭合	断开	亮
闭合	闭合	亮

同样用 0 来表示开关断开或者灯泡不亮，用 1 表示开关闭合或灯泡亮，这个表可以写为如下形式。

上开关	下开关	灯泡
0	0	0
0	1	1
1	0	1
1	1	1

同样的，两个开关可以调换，因此上表也可以写为如下形式。

并联的开关	0	1
0	0	1
1	1	1

你可能已经猜到了，这与布尔代数中的 OR 是一样的。

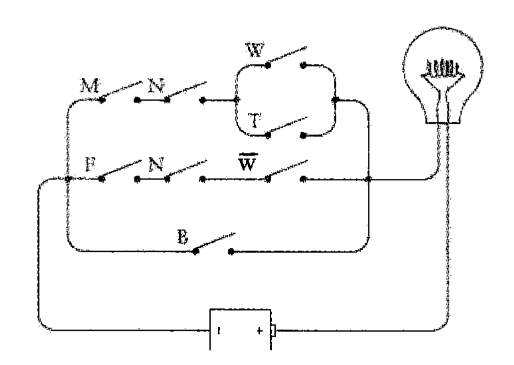
OR	0	1
0	0	1
1	1	1

这就是说，两个开关并联相当于布尔代数中的 OR 运算。

最初当你走进宠物商店的时候，告诉店员：“我想要一只公猫，已绝育的，白色或褐色都可以；或者一只母猫，已绝育的，除了白色任何颜色都可以；或者一只黑猫。”店员会得出以下表达式：

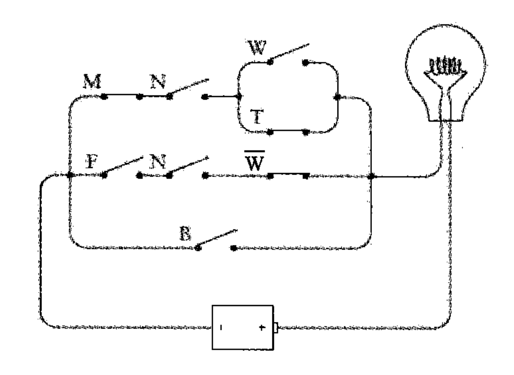
$$(M \times N \times (W + T)) + (F \times N \times (1 - W)) + B$$

既然你知道两个开关串联表示逻辑 AND（用符号“×”表示）；两个开关并联表示逻辑 OR（用符号“+”表示），因此你可以将 8 个开关做如下连接。

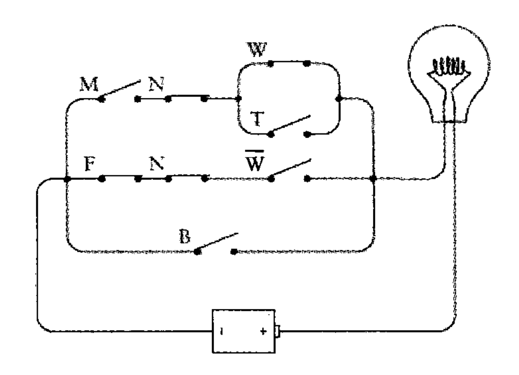


此电路中的每个开关都用一个字母来标记——与在布尔表达式中所用的字母一样（$\overline{W}$ 表示 NOT W，它是 $1-W$ 的另一种表示方式）。如果按照从左到右、从上到下的顺序遍历电路图，你就会以同样次序遇到出现在表达式中的字母。在表达式中每个符号“×”对应电路中的两个开关（或者两组开关）串联的点。在表达式中每个符号“+”对应电路中两个开关（或两组开关）并联的位置。

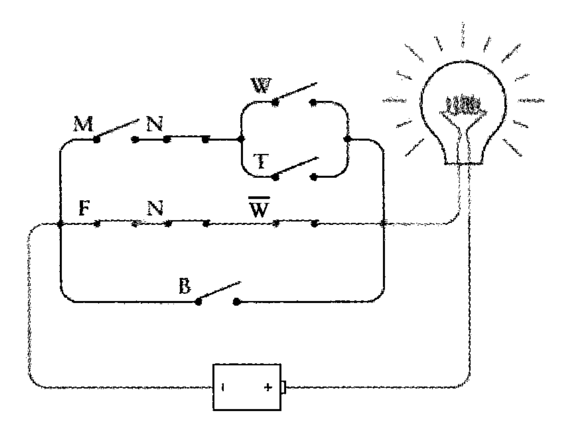
与之前一样，店员首先拿出了一只未绝育的褐色公猫。闭合相应的开关，如下图所示。



尽管开关 ***M***、***T*** 和 ***W*** 闭合了，但是没有成功地点亮灯泡。接着，店员又拿出了一只已绝育的白色母猫。



相应的开关闭合后依然没有点亮灯泡。但是，最后店员拿出了一只已绝育的灰色母猫。



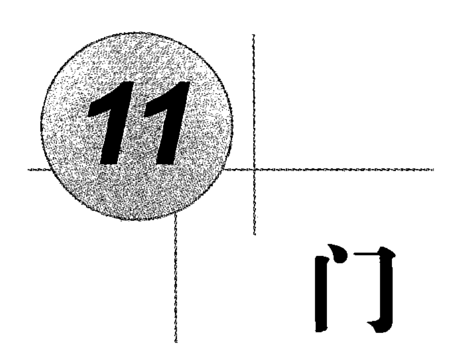
这次，灯泡被成功点亮了，表明这只猫符合你的全部要求。

乔治·布尔并没有连接这样一个电路。他没有兴趣去观察布尔表达式在开关、线路和灯泡中如何实现。当然，这其中存在的一个障碍就是，布尔死后 15 年人类才发明了白炽灯。但是塞缪尔·莫尔斯在 1844 年论证了他的电报机是可行的——早于布尔发表《思维规律的研究》10 年——将电报发声器替换成上述电路中的灯泡本应该是非常简单的。

然而，在 19 世纪，没有人将布尔代数中的 AND 和 OR 同线路中的开关串联及并联关联到一起。没有这样的数学家，没有这样的电学家，没有这样的电报员，没有这样的人。甚至计算机革命的偶像式人物查尔斯·巴贝奇（1792-1871）也没有，他与布尔处在同一时代并且了解布尔的工作，巴贝奇奋斗了一生，他最先设计了差分机（Difference Engine）和分析引擎（Analytic Engine），这些在一个世纪之后都被看做是现代计算机的前

身。本来有些东西可以帮到巴贝奇的，那是什么呢？我们现在知道，那就是根据一台电报器来创建计算机，而非使用齿轮和杠杆来实现计算。

是的，就是电报器。



门

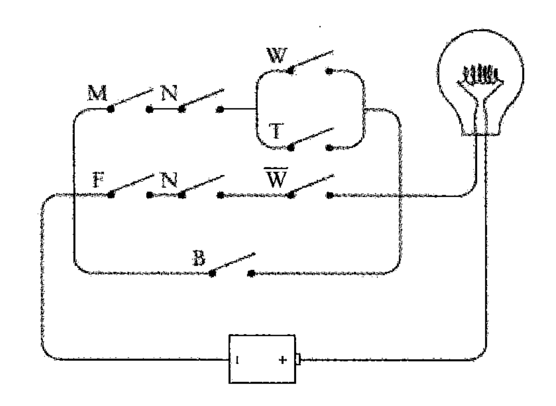
在遥远的未来，关于 20 世纪的早期计算机史早已成为人们印象中模糊的记忆，那时很可能有些人会认为“logic gates”（逻辑门）装置是以著名的微软公司创始人的名字来命名的[1]。其实不然。就像我们所看到的一样，逻辑门与普通的让水通过或者让人通过的门是非常类似的。在逻辑学中，逻辑门的工作方式非常简单——让电流通过或阻止电流通过。

在上一章中，你走进了一家宠物店，并且说：“我想要一只公猫，已绝育的，白色或褐色都可以；或者一只母猫，已绝育的，除了白色任何颜色都可以；或者，只要是一只黑猫就行。”这句话被总结为如下布尔表达式：

$$(\boldsymbol{M} \times \boldsymbol{N} \times (\boldsymbol{W} + \boldsymbol{T})) + (\boldsymbol{F} \times \boldsymbol{N} \times (1 - \boldsymbol{W})) + \boldsymbol{B}$$

同样也可以用这样一个由开关和灯泡组成的电路来表示，如下图所示。

1 Bill Gates 中的 Gates 在英文中有“门”的意思——译者注



这样的电路有时被称为网络（network），而如今这个词更多地被用来描述计算机之间的连接，而不仅指多个开关的集合。

尽管这个电路中的所有元件早在19世纪就都已经被发明出来了，但在那个时代，没有人意识到布尔表达式可以在电路中实现。这个等价关系直到20世纪30年代才被发现。主要贡献人是克洛德·艾尔伍德·香农（生于1916）。1938年，香农在麻省理工学院完成了那篇题为《继电器和开关电路的符号分析》（*A Symbolic Analysis of Relay and Switching Circuits*）的著名硕士论文，在文中阐述了这个问题（10年之后，他又发表了论文“通信的数学原理”，即 *The Mathematical Theory of Communication*，这是第一篇使用“bit”这个词来表示二进制数字的文章）。

1938年以前，人们已经知道，当两个开关串联的时候，要同时闭合它们，电流才能通过；当两个开关并联的时候，闭合其中任何一个都可以使电路连通。但是没有人像香农那样能清晰严谨地阐述：电子工程师可以运用布尔代数的所有工具去设计开关电路。此外，如果你简化了一个描述网络的布尔表达式，那么你也可以简化相应的网络。

例如，你想要的猫可以用下列表达式描述：

$$(\boldsymbol{M} \times \boldsymbol{N} \times (\boldsymbol{W} + \boldsymbol{T})) + (\boldsymbol{F} \times \boldsymbol{N} \times (1 - \boldsymbol{W})) + \boldsymbol{B}$$

利用结合律，我们可以重新排列由AND（×）连接的变量，并将表达式写为如下形式：

$$(\boldsymbol{N} \times \boldsymbol{M} \times (\boldsymbol{W} + \boldsymbol{T})) + (\boldsymbol{N} \times \boldsymbol{F} \times (1 - \boldsymbol{W})) + \boldsymbol{B}$$

为了讲清楚这里是如何变换的，我将定义两个新的符号，$\boldsymbol{X}$ 和 $\boldsymbol{Y}$：

$$X = M \times (W + T)$$
$$Y = F \times (1 - W)$$

现在，描述你想要的小猫的表达式可以写为：

$$(N \times X) + (N \times Y) + B$$

化简后，我们可以将 X 和 Y 表达式代回去。

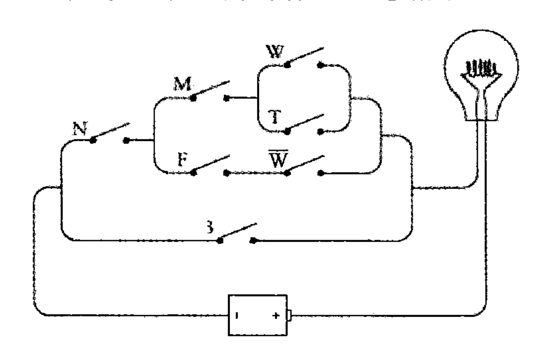
注意，变量 N 在表达式中出现了两次。利用分配率，表达式可以写为只有一个 N 的形式：

$$(N \times (X + Y)) + B$$

现在将 X 和 Y 的表达式带入：

$$(N \times ((M \times (W + T)) + (F \times (1 - W)))) + B$$

由于括号太多，这个表达式看起来并不简单。但是这个表达式中少了一个变量，也就意味着在网络中少了一个开关。以下就是简化后的电路图。

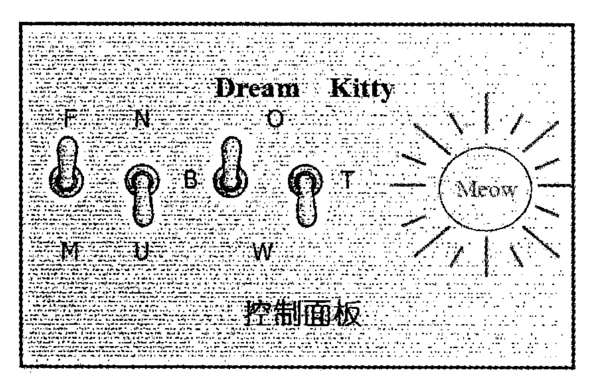


确实，证明修改前后的两个电路功能等价要比去证明两个表达式功能相同简单许多。

事实上，网络中仍然有三个开关是多余的。理论上讲，你只需要四个开关来定义你心目中的猫咪即可。这是为什么呢？每个开关都是一个二进制数。你可以设定一个开关代表猫的性别（开关断开表示是公的，而闭合表示是母的），另一个开关闭合表示猫是否有生育能力（开关断开表示未绝育，而闭合则表示已绝育）。另两个开关用来表示猫的颜色。这里有四种可能出现的颜色（白色、黑色、褐色和其他颜色），而我们知道，四种选择可以用两个二进制位来定义，于是你需要的就是两个颜色开关。例如，两个开关同时

断开表示白色，一个闭合表示黑色，另一个闭合表示褐色，同时闭合表示其他颜色。

现在让我们制作一个用来选猫咪的控制面板。这个控制面板上只有四个开关（与你家墙上控制灯开闭的开关很相似），此外面板上还安装了一个灯泡，如下图所示。



开关打到上面是指开关闭合，反之是指开关断开。或许用来表示猫咪颜色的两个开关标记有点难于理解，这是为了将面板做得简洁而不得已导致的一个小缺憾。在表示颜色的两个开关中，左边的叫做 B，意思是说只要它闭合（如上图所示）就表示黑色。两个开关中右边的那个叫做 T，意思是说只要它闭合的时候就表示褐色。如果两个开关都闭合则表示其他颜色，这个选择叫做 O。两个开关都断开的时候表示白色，用 W 表示，字母写在下部。

在计算机术语中，开关是一种输入设备（input device），输入是控制电路如何工作的信息。在本例中，输入开关对应于 4 个二进制数信息，这些信息用来描述一只猫。输出设备（output device）就是灯泡。如果开关描述了一只符合标准的猫，灯泡就会亮。比如上面控制面板所示的开关就表示了一只未绝育的黑色母猫。这只猫符合你的标准，因此灯泡是亮的。

现在我们要做的就是设计一个电路来控制这个面板工作。

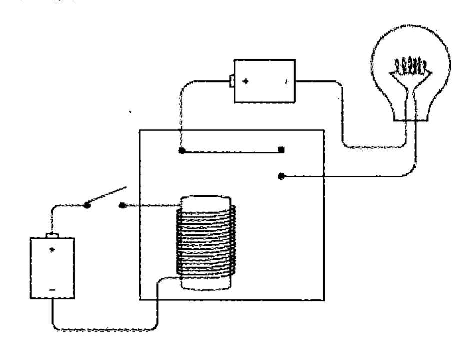
前面提到，香农的论文题目是《继电器与开关电路的符号分析》。这里的继电器与之前我们在第 6 章提到的电报系统中的继电器很类似。然而在香农发表论文的时候，继电器已经被广泛地用于其他领域，特别是在庞大的电话系统网络中。

继电器像开关一样，可以串联或并联在电路中执行简单的逻辑任务。这种继电器的组合叫做逻辑门（logic gates）。这里提到的逻辑门执行“简单”逻辑任务是指逻辑门只完

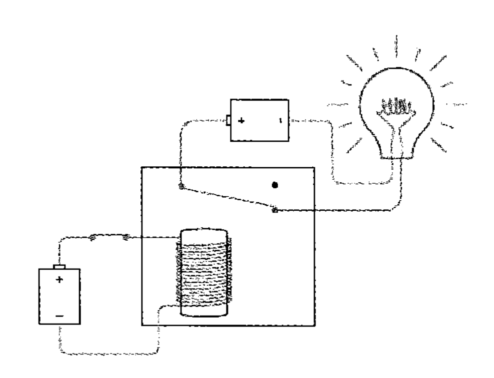
成最基本的功能。继电器优于开关之处就在于，继电器可以被其他继电器所控制，而不必由人工控制。这就意味着，这些简单的逻辑门组合起来可以实现更复杂的功能，例如一些简单的算术操作。事实上，下一章中，我们就将介绍如何利用电线、开关、灯泡、电池和电报继电器来制作一个加法器（尽管它只能用于二进制数计算）。

前面提到过，继电器对于电报系统的工作而言是至关重要的。在长距离情况下，连接电报站的电线具有很高的电阻。这就需要采取一些措施来接收微弱信号并把它增强后再发射出去。继电器就是通过电磁铁控制开关来实现这一目的的。实际上，继电器是通过放大微弱信号来生成强信号的。

就我们的目的而言，我们对于继电器放大微弱信号的功能并不感兴趣。我们真正感兴趣的是继电器可以作为一个电流控制而非人工控制的开关。我们可以将继电器、开关、灯泡和两节电池按照下图连接。



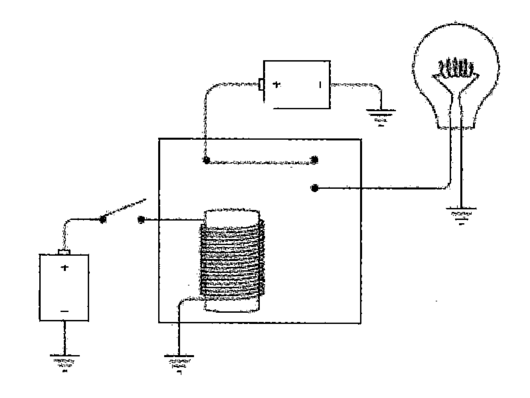
注意，左边的开关是断开的，灯泡没有发光。当闭合开关的时候，左端电池产生电流流过缠在铁芯上的圆线圈。于是铁芯产生了磁性，将上面的弹性金属簧片拉下，使回路接通，灯泡发光。



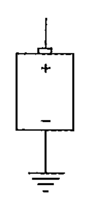
当电磁铁将金属簧片拉下来时，我们称继电器被“触发”(triggered)。当左边的开关断开的时候，铁芯的磁性消失，金属簧片回到原位。

这看起来似乎是一种间接控制灯泡发光的方法，而实际上它就是这样的。如果我们只关心点亮开关，完全可以将继电器省略掉，但我们所关心的并不仅仅是点亮灯泡。我们还有更宏大的目标。

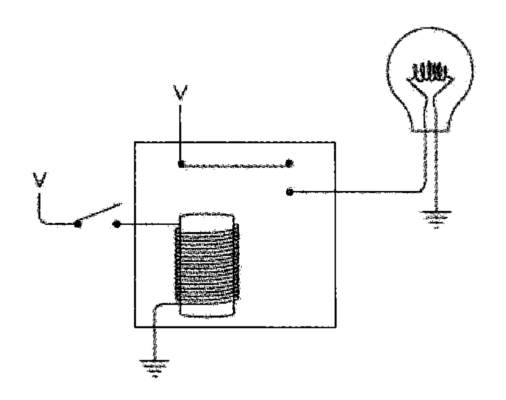
在这一章中，(在逻辑门建立之前)我们将多次运用继电器，因此就需要将上面那幅电路图简化。我们可以利用接地的方式减少一些电线。在这种情况下，大地仅代表了一个公共端，并不是真正意义上的物理接地。



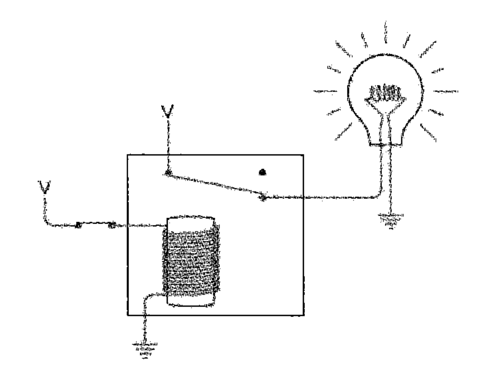
这看起来并不像简化的结果，但到现在我们还没有完成。注意两节电池的负极都是接地的。因此，我们可以看到这样的符号。



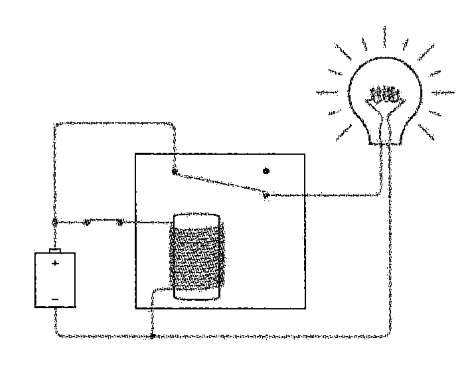
和第5章、第6章中一样，可以用大写字母V(代表电压)来代替上图中的电池。这样，继电器看上去就如下图所示。



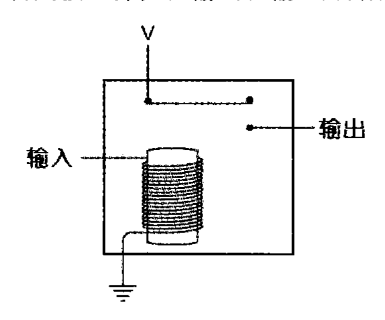
开关闭合，电流从 V 端流出，经过电磁铁芯流到大地上。产生磁效应吸合金属簧片，从而连通了 V、灯泡和地之间的电路，使灯泡发光。



上面的图显示了两个电源和两个接地端。但是在这一章的所有电路中，所有电源，即“V”是可以彼此互连的，接地端也如此。在本章和下一章的电路中，所有继电器网络和逻辑门只需要一节电池，尽管可能是一节很大容量的电池。例如，上述电路图可以画为如下只用一节电池的方式。

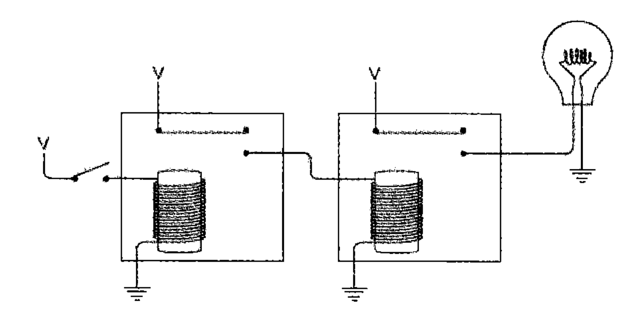


但是，这个电路图中没有清楚地表明，为什么我们要使用继电器。可以先不看回路，单单看继电器，像前面的控制面板一样，从输出和输入开始。

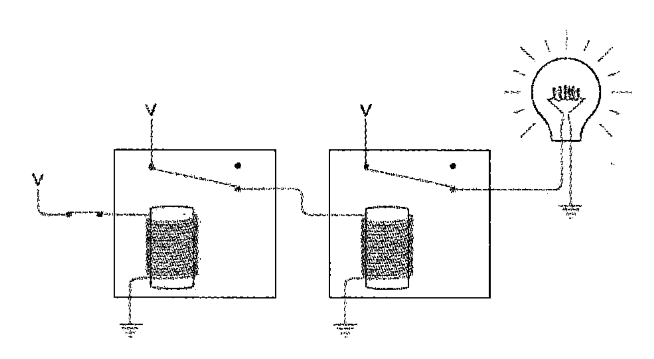


当电流流经输入时（例如，用一个开关把输入连到“V”端），电磁铁就会被触发，输出就得到一个电压。

继电器的输入不一定只能是开关，其输出也未必只限于灯泡。一个继电器的输出可以连到另一个继电器的输入，例如下面这个电路：

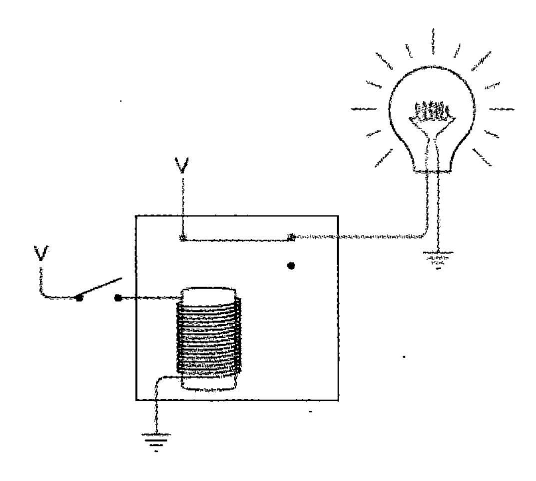


当开关闭合，第一个继电器被触发，提供电压给第二个继电器。于是第二个继电器被触发，使灯泡发光。

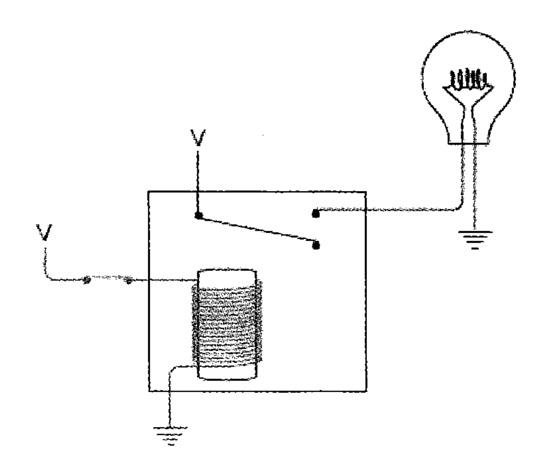


连接继电器是建立逻辑门的关键。

实际上，灯泡可以采用两种方式连接到继电器上。注意，弹性金属簧片是被电磁铁拉下来的。平时，金属簧片与上端触点相接触，当电磁铁拉动它时，它就会与下端触点相接触。我们之前一直把金属簧片与下方触点接触作为继电器的输出，而其实也可以把它与上方触点相接触作为输出。当我们这样使用的时候，继电器的输出恰好相反，当输入开关断开的时候，灯泡发光。



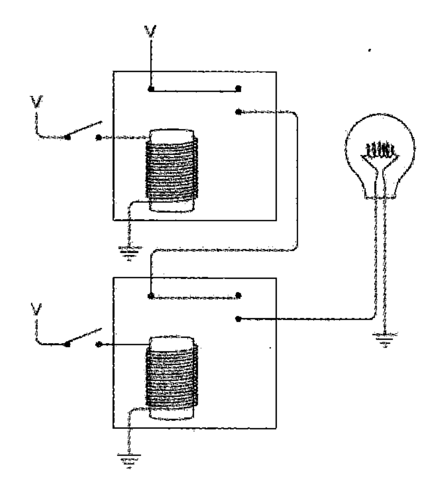
当输入开关闭合时，灯泡熄灭。



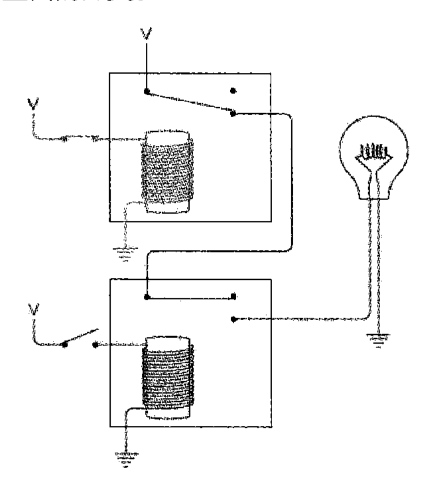
用开关术语来说，这种继电器叫做双掷继电器。它拥有两个输出，但是这两个输出在电的极性上是对立的——当一端有电压时，另一端就没有。

另外，如果想象不出现代继电器是什么样子，你可以在你们当地电器行的简易透明包装中看到一些。有些继电器就像方形小冰块一样大，如零件型号为 275-260 和 275-214 的继电器就是这种大小的继电器，而且它们也十分耐用。这些继电器内部的元件被封在一个干净的塑料外壳里，因此你可以看到电磁铁和金属簧片。本章和下一章所描述的电路都使用的是元件号为 275-240 的继电器，这种继电器体积小（大约一块口香糖大小）而且价格便宜（每个 2.99 美分）。

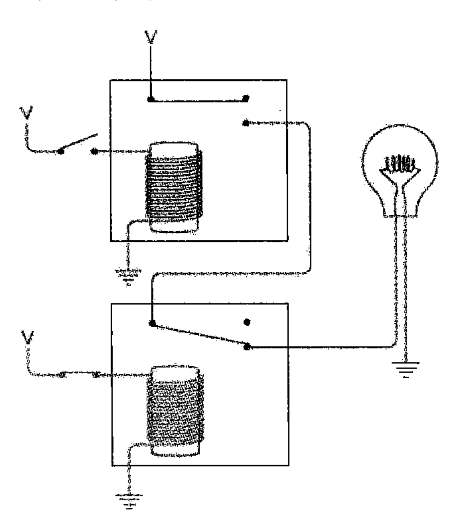
前面提到，两个开关可以串联，同样地，两个继电器也可以串联。



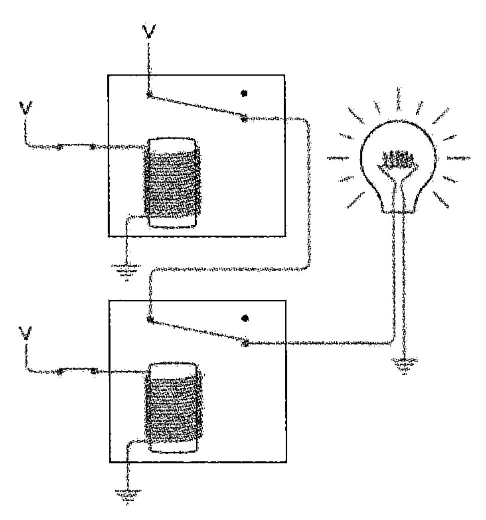
上面继电器的输出为下面继电器提供电压。如上图所示，当两个开关都断开的时候，灯泡不发光。我们先闭合上面的开关。



灯泡仍然不亮，因为下面的开关一直是断开的，这个继电器没有被触发。我们现在断开上面的开关，并闭合下面的开关。



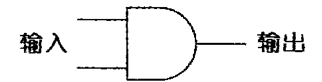
灯泡仍然不亮。由于上面的继电器没有被触发，电流无法流过灯泡。只有两个开关都闭合的时候灯泡才会被点亮。



这时，两个继电器都被触发，电流从 V 流经灯泡后流入到地中。

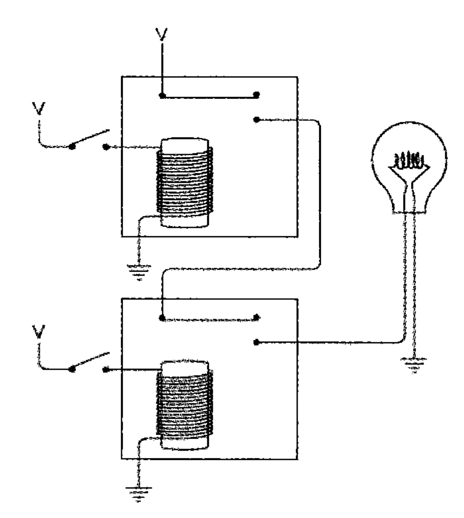
就像两个开关串联一样，这两个继电器也执行了逻辑操作。只有当两个继电器都被触发的时候灯泡才会亮。这样两个继电器的串联被称为一个“与门”。为了避免复杂的图

示，电气工程师用如下专门的符号表示一个与门。

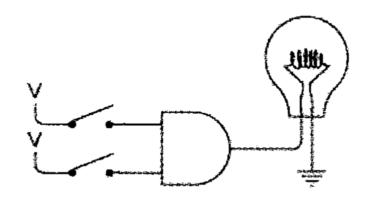


这是四个基本逻辑门中的一个。与门有两个输入端（上图中的左端）和一个输出端（上图中的右端）。这样表示的与门通常输入在左边，输出在右边。这是因为人们习惯于由左向右的阅读方式，在读电路图的时候也会由左向右。但与门也可以画成输入在上端、右端或者下端。

有两个继电器、两个开关和一个灯泡的原始电路图如下所示。

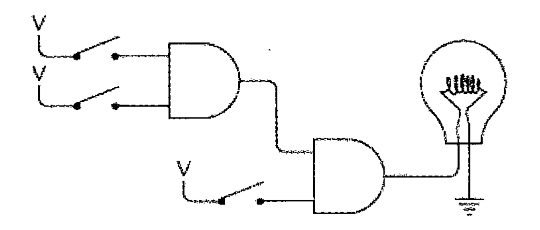


使用“与门”符号，上图可以画为如下所示的图。



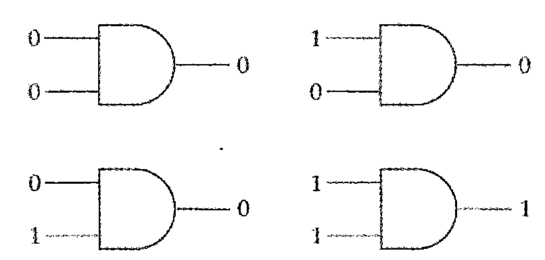
注意，与门的符号不仅仅代替了两个串联的继电器，而且还暗示着上面的继电器与电源相连，两个继电器都接地。只有当上面的开关与下面的开关都闭合的时候，灯泡才会发光。这就是称它为“与门”的原因。

与门的输入未必一定要和开关相连，而且输出也不一定只能与灯泡相连。我们真正要处理的是输入端的电压和输出端的电压。例如，一个与门的输出可以作为另一个与门的输入，如下所示。



只有当三个开关全部闭合的时候，灯泡才会亮。只有当上面两个开关全闭合的时候，第一个与门的输出才会触发第二个与门中的第一个继电器。而最下面的开关闭合会触发第二个与门中的第二个触发器。

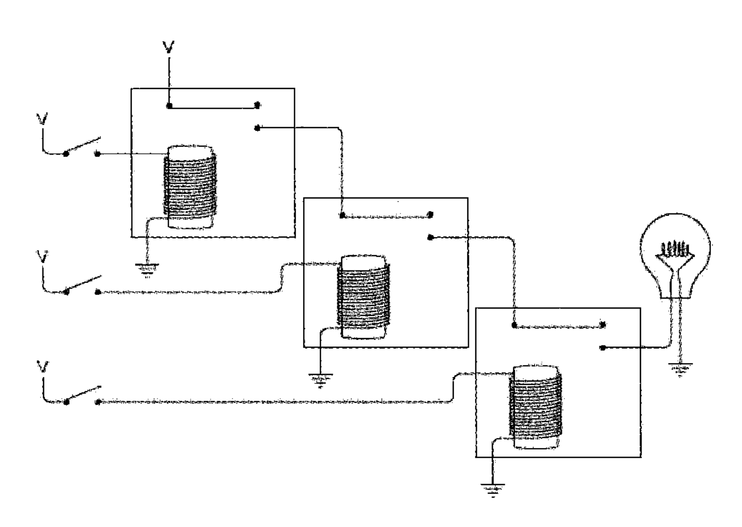
如果我们将低电平视为 0，将高电平视为 1，那么与门的输入和输出之间的关系如下所示。



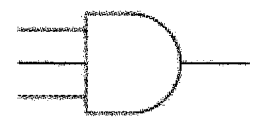
像两个开关串联一样，与门的输入与输出之间的关系同样可用下表来描述。

AND	0	1
0	0	0
1	0	1

同样可以让与门有多个输入端。例如，将三个继电器串联，如下图所示。

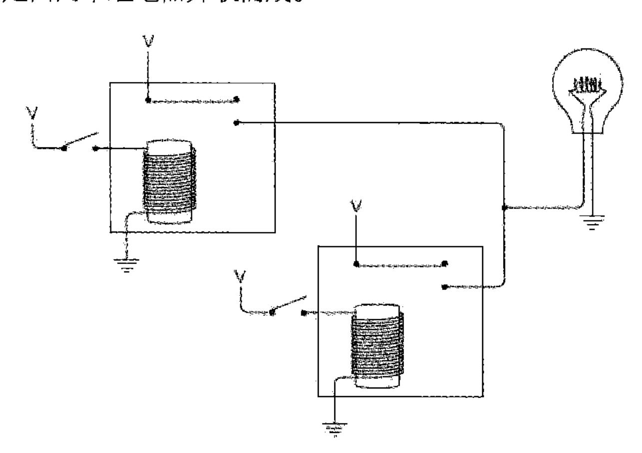


只有当三个开关全部闭合时灯泡才会发光。这个结构可以用如下符号来表示。

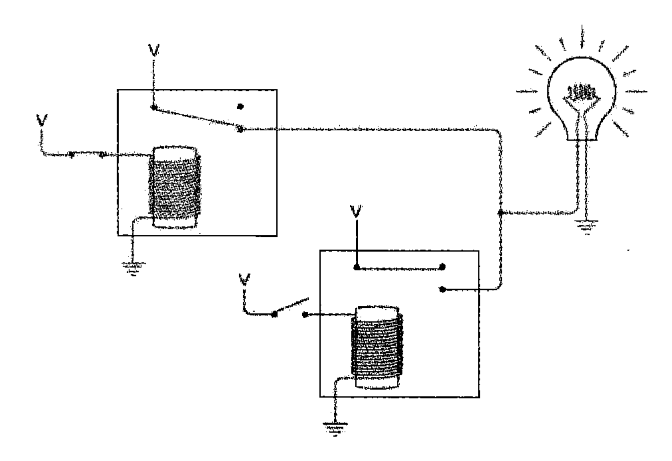


它被称为三输入端与门。

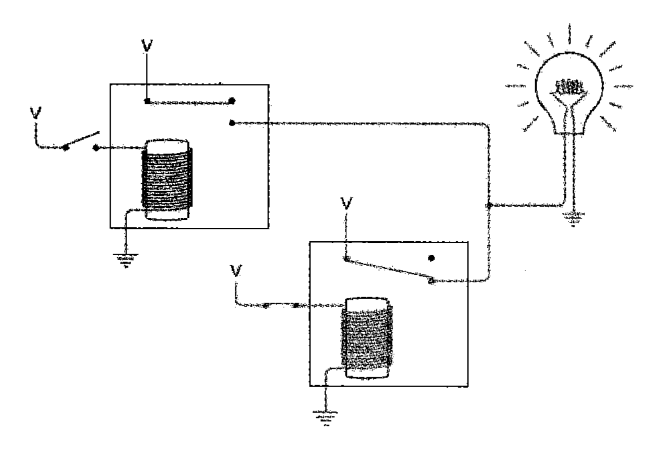
以下逻辑门是由两个继电器并联而成。



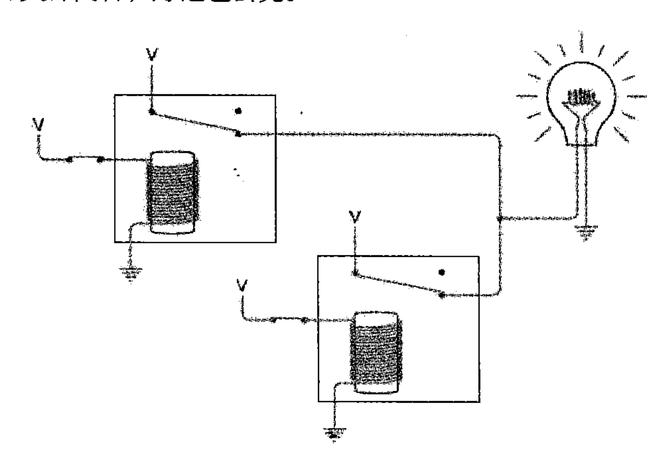
注意两个继电器的输出是接在一起的，这个连在一起的输出为灯泡提供了电源。任何一个继电器都能点亮灯泡。例如，如果闭合上面的开关，灯泡会亮。这时，灯泡从左边的继电器得到了电源供应。



同样的，如果我们将上面的开关断开，并闭合下面的开关，灯泡也会发光。



如果两个开关都闭合，灯泡也会亮。

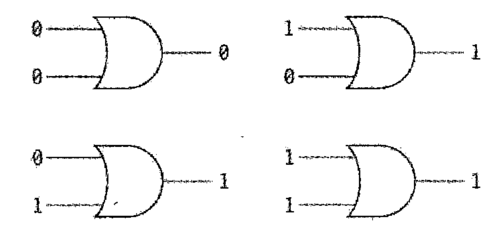


显而易见，当上面的开关或下面的开关闭合，灯泡都会发光，这里的关键词是“或”，因此这样的门被称为“或门”。电气工程师用如下符号表示或门。

输入 输出

它和与门的符号稍微有点相似，但是输入端的一边是弧线，像英文单词“OR”中字母“O”一样（这样可以帮你分清它们）。

或门的两个输入中，只要有一个加上电压，输出就是高电平。如果将低电平看做 0，高电平看做 1，那么或门也有四种可能的组合状态。

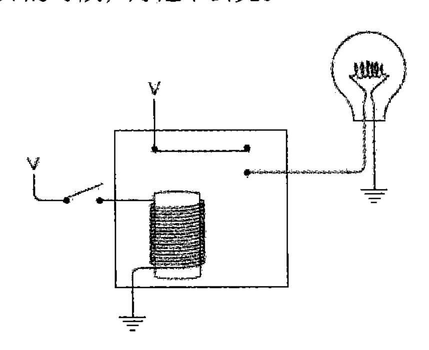


像总结与门一样，我们可以将或门的输入和输出关系总结为一个表，如下所示。

OR	0	1
0	0	1
1	1	1

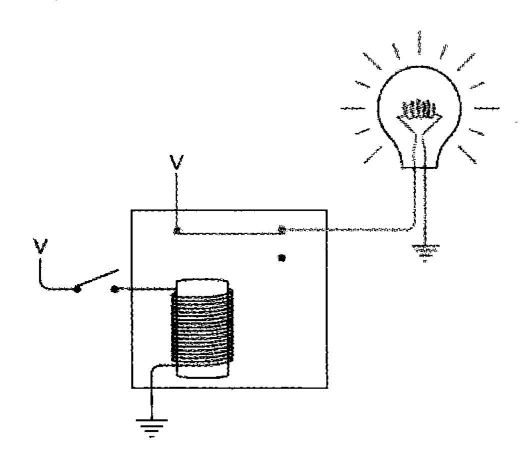
或门也可以有多个输入端（只要任一个输入端为 1，其输出端就为 1，只有所有的输入端都为 0 时，输出端才为 0）。

前面解释了为什么我们所用的继电器叫双掷继电器，这是因为其输出有两种不同的连接方式。通常在开关断开的时候，灯泡不会亮。



当开关闭合时，灯泡发光。

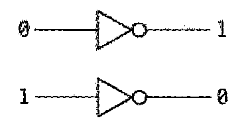
也可以用另外一种连接方式，使开关断开时灯泡被点亮。



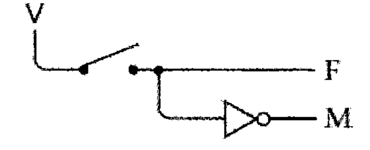
这样的话，开关闭合，灯泡就会熄灭。以这种方式连接的继电器叫做反向器(inverter)。反向器不是逻辑门（一个逻辑门通常有两个或多个输入），尽管如此，它的用处还是很广。反向器可以用如下的专门符号来表示。

输入 ——▷○—— 输出

由于它能将 0（低电平）转换为 1（高电平），因此被称为反向器，反过来也是一样的。

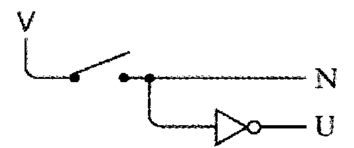


利用反向器、与门和或门，我们就可以着手去设计一个自动选择理想猫咪的控制面板了。首先从开关开始，第一个开关闭合表示母猫，断开表示公猫。因此，我们可以得到两个信号，把它们分别叫做 F 和 M，如下图所示。

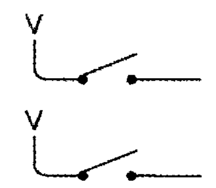


当 F 为 1 时，M 为 0，反之亦然。同样，第二个开关闭合表示这只猫已绝育，断开则

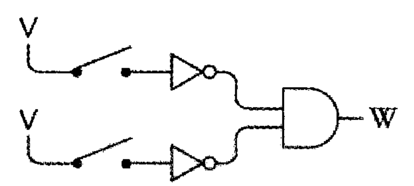
表示这只猫未绝育。



下面两个开关更加复杂。两个开关的不同组合分别表示四种不同的颜色。以下为两个接有电源的开关。

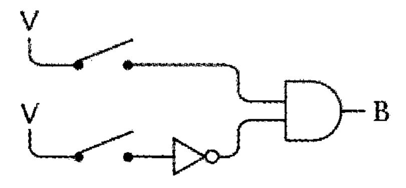


当两个开关都断开时（如上图所示），表示白色。以下就是如何运用两个反向器和一个与门来得到 W 信号的方式。如果你选择一只白猫，W 就为高电平（1）；否则，就为低电平（0）。

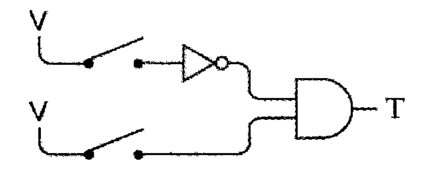


当两个开关断开时，两个反向器的输入均为 0，两个反向器的输出（也就是与门的输入）都为 1。这就意味着与门的输出为 1。如果有一个开关闭合，与门的输出就为 0。

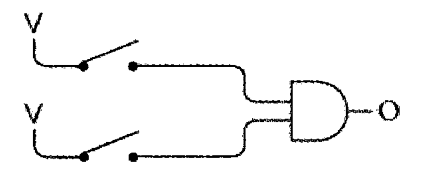
要想用闭合第一个开关表示一只黑猫，可以利用一个反向器和一个与门实现。



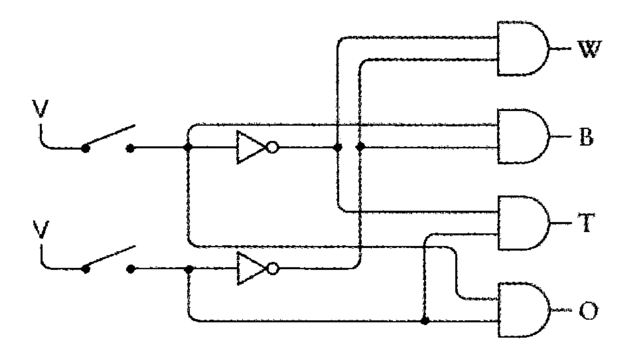
只有当第一个开关闭合而第二个开关断开的时候，与门输出才为 1。同样，如果第二个开关闭合而第一个开关断开，与门的输出也为 1，我们用此来表示褐色的猫。



如果两个开关同时闭合则表示其他颜色的猫。



现在，我们将四个小电路合并成一个大电路（按照惯例，黑实心点表示交叉线之间是连接的，没有黑实心点的交叉线则表示仅仅是穿过，没有连接）。



这个电路看起来非常复杂，但是如果你仔细地沿着线路走，看清每个与门的输入是从哪来的，而暂不论这些输入去向何方，电路的工作原理就会一目了然。如果两个开关都断开，输出信号 W 就为 1，其他都为 0。如果第一个开关闭合，则输出信号 B 为 1，其他为 0，依此类推。

在连接门和反向器的时候有一些规则，影响它们的连接方式：一个门（或反向器）的输出可以作为一个或多个其他门（或反向器）的输入。但是两个或多个门（或反向器）的输出是不可以相互连接的。

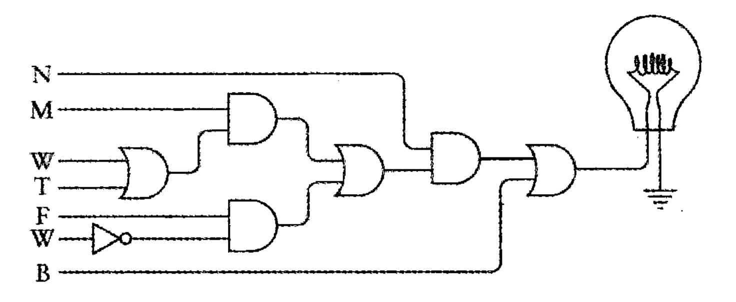
这个由 4 个与门和 2 个反向器连接成的电路叫做“2-4 译码器”。输入为 2 个二进制位，各种组合共表示 4 个不同的值。输出是 4 个信号，任何时刻只能有一个是 1，至于哪一个是 1 取决于两个输入。利用同样的原理，我们可以构造出 3-8 译码器或者 4-16 译码

器，等等。

简化过的选择猫咪的表达式为：

$$(N \times ((M \times (W + T)) + (F \times (1 - W)))) + B$$

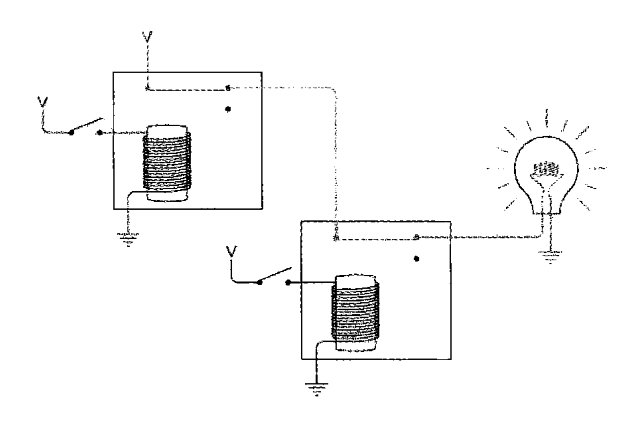
在这个表达式中，每个加号（+），必定对应电路中的一个或门。对于每一个乘号（×），则对应一个与门，电路图如下所示。



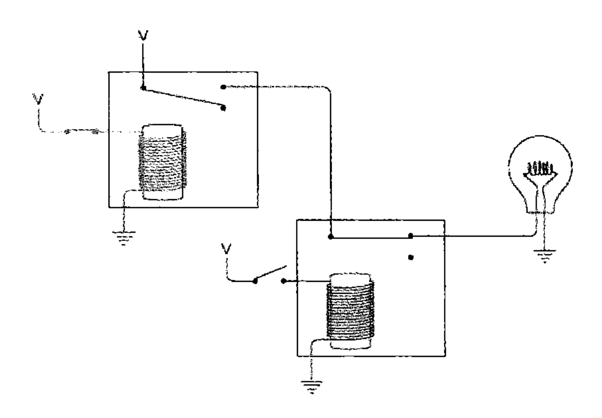
电路图左侧的字母由上到下的顺序与它们表达式中出现的顺序一样。这些信号来源于与反向器相连的开关和 2-4 译码器的输出。请注意用来表示表达式中（1−W）部分的反向器的用法。

这时，你可能会说："这一堆继电器太多了！"，是的，确实如此。每个与门和或门中有两个继电器，一个反向器中有一个继电器。但是我在这里要说的就是你必须习惯它。之后的章节中我们还会用到更多的继电器。还好你不用真的买来并在家中自己连接它们。

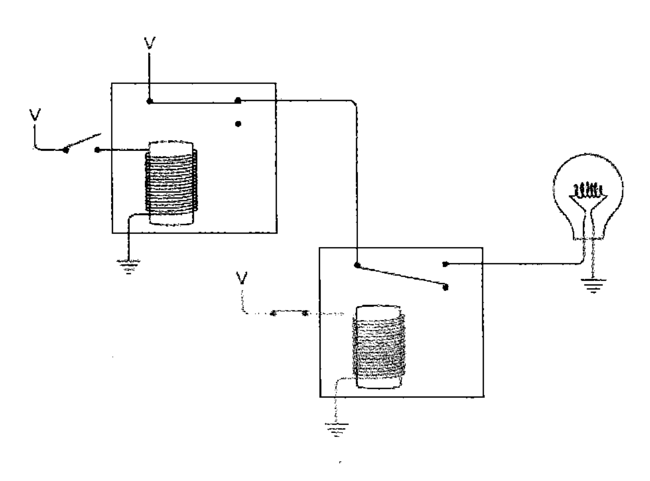
下面要介绍另外两种本章要用到的逻辑门。这两个门都会用到这样一个继电器，该继电器在未被触发时，其输出为高电平（这是用在反向器中的输出）。例如，在下面这种配置中，第一个继电器的输出为第二个继电器提供电源。当两个继电器全都断开时，灯泡发光。



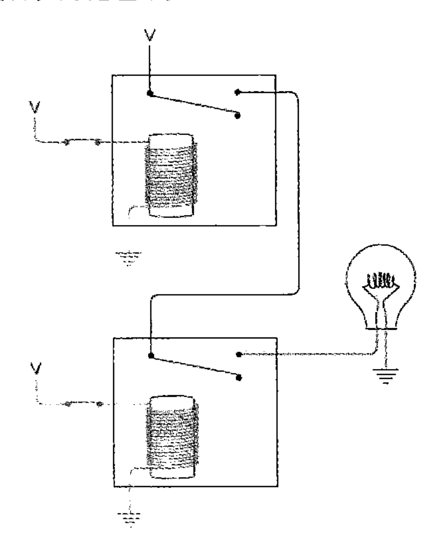
如果上面的开关闭合，灯泡就会熄灭。



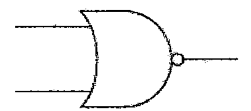
灯泡熄灭是由于第二个继电器没有电源供应。同样的，如果下面的开关闭合，灯泡也会熄灭。



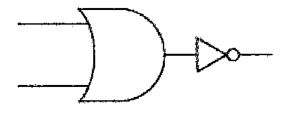
如果两个开关全部闭合，灯泡也不亮。



这些结果恰恰与或门相反，这个门称为“或非门”，简称 NOR，用以下符号表示。



除去输出部分的小圆圈，这个符号与或门非常相像。小圆圈表示“反向”，所以或非门也可用下面的符号表示。

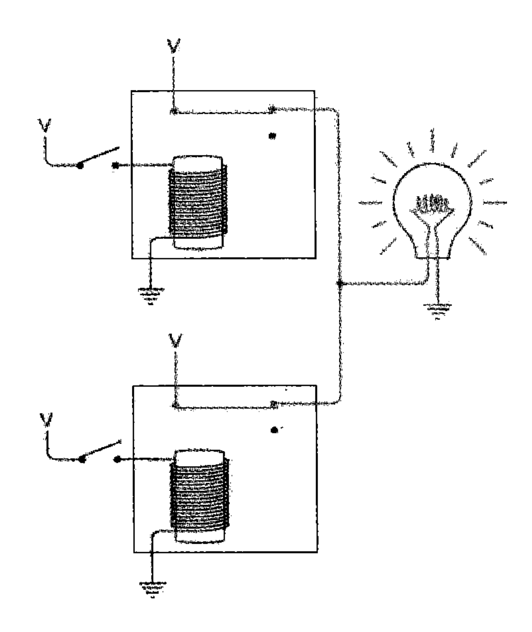


或非门的输出如下表所示。

NOR	0	1
0	1	0
1	0	0

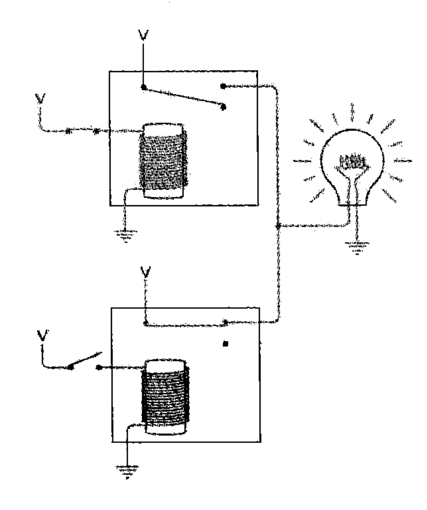
这个表所显示的结果与或门正好相反，在或门中，两个输入中有一个为 1 输出就为 1，只有两个输入都为 0，输出才为 0。

下面是另一种连接两个继电器的方法。

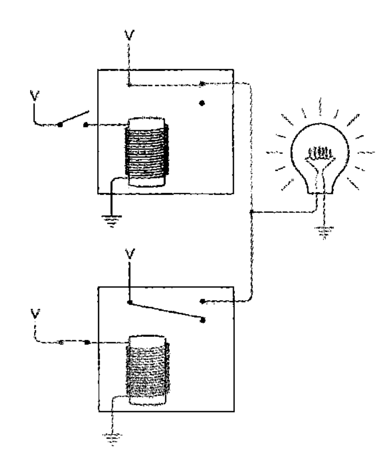


在这种情况下，两个输出连接在一起，与或门的布局类似，但是却采用了另一种输出接法。灯泡在两个开关全断开时被点亮。

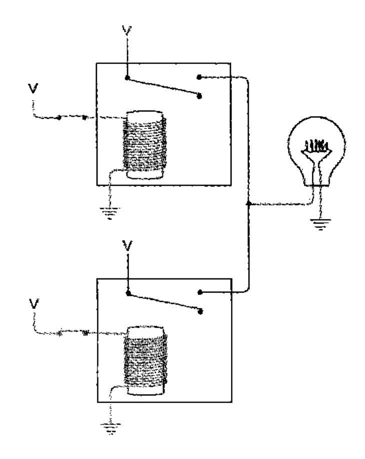
当上面的开关闭合时，灯泡依然是亮的。



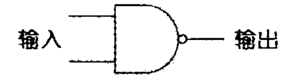
同样，当只有下面的开关闭合时，灯泡也依然是亮的。



只有当两个开关全闭合的时候，灯泡才会熄灭。



这一结果和与门恰恰相反。这种逻辑门被称为与非门，或简称 NAND。与非门的符号和与门类似，但在输出部位多了一个小圆圈，意思是输出和与门正好相反。



与非门的输出如下表所示。

NAND	0	1
0	1	1
1	1	0

注意与非门的输出是和与门完全相反的。与门只有当输入全为 1 的时候输出才为 1，否则输出就为 0。

到此为止，我们已经看到可以用四种不同的方式来连接有两个输入、一个输出的继电器，每一种方式的行为功能都不一样。为了避免重复画继电器，我们将它们称为逻辑门，并用电气工程师们所使用的专门符号来表示它们。特定逻辑门的输出取决于它的输入，输出与输入的关系可总结为以下几个表格。

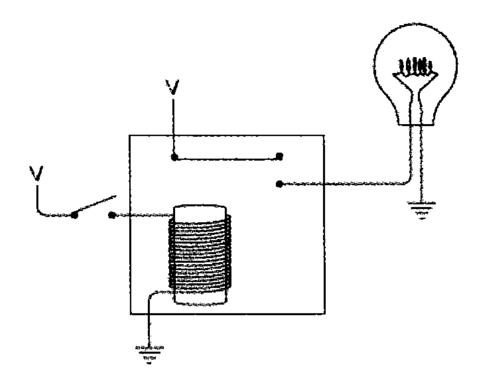
AND	**0**	**1**
0	0	0
1	0	1

OR	**0**	**1**
0	0	1
1	1	1

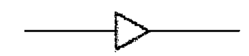
NAND	**0**	**1**
0	1	1
1	1	0

NOR	**0**	**1**
0	1	0
1	0	0

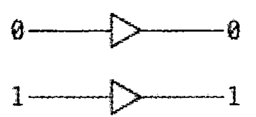
现在，我们已经有四个逻辑门和一个反向器。把这些工具组合到一起其实就是原始的继电器，如下图所示。



这叫做缓冲器（buffer），可用如下符号表示。



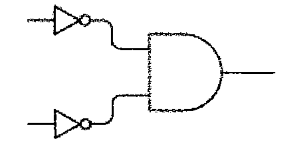
除了输入端的小圆圈，这个符号与反向器是很相似的。很明显，缓冲器"没有什么作用"，它的输入与输出是相同的。



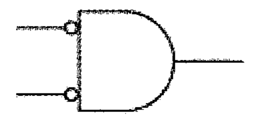
但是在输入信号很微弱的时候，缓冲器就可以派上用场。之前提到过，这也就是很多年前在电报机中使用继电器的原因。另外，缓冲器还可以用于延迟一个信号。这是因为继电器需要一点时间——几分之一秒——才会被触发。

本书从这里开始，继电器将会极少出现了。取而代之地，以后的电路会由缓冲器、反向器、四种基本逻辑门和其他由逻辑门组成的复杂电路（如 2-4 译码器）组成。当然，所有这些器件也是由继电器构成的，但我们用不着直接使用它了。

前面在建立 2-4 译码器的时候，曾出现过这样一个小电路。

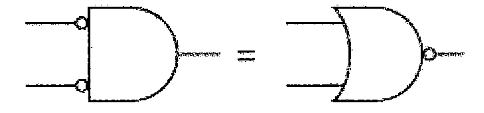


两个输入信号经过反向器后作为与门的输入。这样的组合有时可以去掉反向器而画成如下的形式。



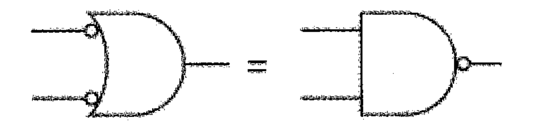
注意与门输入端的小圆圈。这些小圆圈表示信号在那一点被反转——将 0（低电平）转换为 1（高电平），反之亦然。

实际上，带有两个反向输入的与门和或非门是等价的。



只有当输入都为 0 的时候，其输出才为 1。

类似的，带有两个反向输入的或门和与非门也是等价的。



只有当输入全为 1 时，输出才为 0。

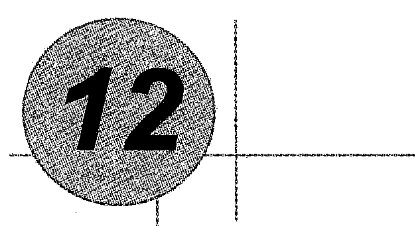
这两组等价关系就是摩根定律在电路中的实现。摩根是维多利亚女王时代的另一位数学家，比布尔大 9 岁，于 1847 年发表《形式逻辑》（*Formal Logic*）一书，与布尔的《逻辑的数学分析》（*The Mathematical Analysis of Logic*）恰好同一天出版（据说是同一天）。实际上，布尔研究逻辑的灵感正源于一场公开的争论，这个争论就发生在摩根和另一个被指剽窃的英国数学家之间（历史最后证实摩根是清白的）。摩根很早就发现了布尔的洞察力。他无私地鼓励布尔，并帮助他开展研究，而今天除了他的这个著名定律之外，他几乎已被人们所遗忘。

摩根定律可以简单地表示为如下形式：

$$\overline{A} \times \overline{B} = \overline{A + B}$$
$$\overline{A} + \overline{B} = \overline{A \times B}$$

A 和 B 是两个布尔操作数。在第一个表达式中，两个操作数先被取反，再进行与运算。这与两个操作数进行或运算后再取反（即或非）的结果是一样的。在第二个表达式中，两个操作数先取反，再进行或运算，这和两个操作数先做与运算后再取反（即与非）的结果是等价的。

摩根定律是简化布尔表达式的一种重要手段，因此也可以用来简化电路。从历史的角度来说，这正是香农的论文带给电气工程师们的真正意义。但在本书中简化电路不是重点，我们关注的是让一切有效地运转而不是以最简的形式运转。我们下面要做的就是一台加法器。



二进制加法器

加法是算术运算中最基本的运算，因此如果想搭建一台计算机（这也正是本书所隐含的内容），那么首先就要造出可以计算两个数的和的器件。当你真正面对它时，就会发现，原来加法计算就是计算机要做的唯一工作。如果我们可以造出加法器，同样地，就可以利用加法来实现减法、乘法和除法，计算按揭付款，引导火箭飞到火星、下棋，以及填写我们的话费账单。

这一章中我们要创建的加法器与现代的计算器和计算机来比，将会很庞大、很笨拙、很慢，而且运转起来噪声不断。最有趣的是我们用来制作加法器的全部零件，都是像开关、灯泡、导线、电池、逻辑门、已经预先连接在各种逻辑门中的继电器等这些在之前的章节中学过的非常简单的电子器件。这个加法器所包含的所有器件早在 120 多年前就已经被发明出来了。我们并不用在房间中实际搭建出什么，相反地，可以在纸上以及我们的头脑中来搭建这个加法器。

这个加法器完全用于二进制数的计算，而且没有现代加法器那么便利。你不能使用键盘来给出需要相加的两个数，相反所要用到的是一排开关。计算结果是通过一排灯泡来显示的，而非以数字的形式来显示。

但是，这个加法器确实可以将两个数相加，其方法与计算机计算加法的方式非常相似。

计算二进制数加法与计算十进制数加法非常相似。如果你想要让 245 和 673 这两个十进制数相加，你会把这个问题分解为几个简单的步骤。每个步骤只需要将两个十进制数相加。在这个例子中，首先要将 5 和 3 相加。生活中，记住加法表能更快地解决问题。

二进制数加法与十进制数加法最大的不同就在于二进制数加法中用到了一个更为简单的加法表。

+	0	1
0	0	1
1	1	10

如果你是一头鲸鱼并且在学校中学习了这个表格，你会大声说出：

0 加 0 等于 0；
0 加 1 等于 1；
1 加 0 等于 1；
1 加 1 等于 0，进位为 1。

以上加法表可以重新写为下面带有前导零的形式，这样每个结果就都是一个 2 位的值。

+	0	1
0	00	01
1	01	10

像这样，一对二进制数相加的结果中具有两个数位，其中一位叫做加法位（sum bit），另一位则叫做进位位（carry bit，例如，1 加 1 等于 0，进位为 1）。下面我们将二进制数加法表分成两个表格，第一个是表示加法的。

+加法	0	1
0	0	1
1	1	0

第二个是表示进位的。

+进位	0	1
0	0	0
1	0	1

用这种方法来看二进制加法非常方便，因为在我们的加法器中加法与进位是分别进行的。搭建一个二进制加法器需要我们首先设计一个电路，通过该电路执行这些操作。完全用二进制数，问题将在很大程度上得以简化，因为，电路中的所有器件，像开关、灯泡、导线等都可以用来表示二进制数的位。

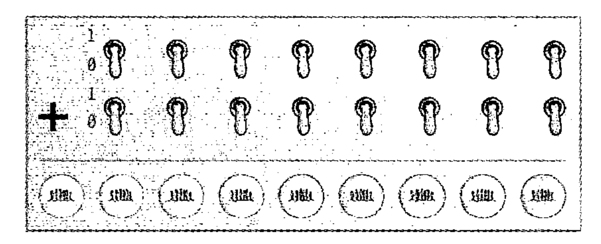
与十进制数加法一样，我们将两个二进制数字串由右向左依次逐列相加。

$$\begin{array}{r} 01100101 \\ +\ 10110110 \\ \hline 100011011 \end{array}$$

注意，在我们将从右边数的第 3 列的数相加的时候，会有一个 1 进位到下一列中。这在第 6、第 7、第 8 列也是一样的。

想要让多大的二进制数相加？由于仅仅在头脑中来搭建加法器，所我们可以搭建出一个能够让非常长的二进制数相加的加法器。但是经过理智的考虑之后，我们决定让搭建起来的二进制加法器最高能够执行的加法长度为 8 位。也就是说，我们想要相加的二进制数，其范围是从 0000-0000 到 1111-1111，即十进制数的 0 到 255。两个 8 位二进制的和最大可为 1-1111-1110，即 510。

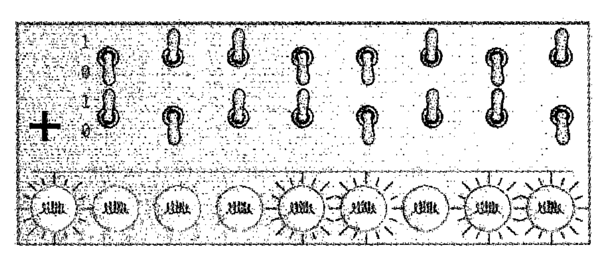
加法器的控制面板如下图所示。



在这个面板中，有两排开关，每排 8 个。这些开关就是输入设备，我们将用它们来“输入”两个 8 位二进制数。在这套输入设备中，开关断开（关）即表示 0，闭合（开）表示 1，这与你房间墙上的开关是一样的。面板底部的输出设备是一排灯泡，共 9 个。这些灯泡用来显示结果。不发光的灯泡表示 0，发光的灯泡表示 1。这里有 9 个灯泡，因为两个

8 位二进制数的相加结果可能是一个 9 位的二进制数。

加法器的其他部分是以各种形式连接起来的逻辑门。开关将触发逻辑门中的继电器来点亮相应的灯泡。例如，如果我们要将 0110-0101 和 1011-0110（之前例子中的两个数）相加，则要将相应的开关置于下图所示位置。



灯泡发光显示结果为：1-0001-1011（然而，这只是一个希望的结果，因为我们还没有将加法器真正搭建出来！）

在上一章中提到过，本书中我们将会用到许多继电器。这个 8 位二进制加法器中所用到的继电器不少于 144 个，其中我们用来相加的 8 对二进制位，每对都需要 18 个继电器。如果将全部电路展示出来，你一定会崩溃的。没有人能看懂以各种方式连接起来的 144 个继电器所表达的意义。相反地，我们要利用逻辑门来分阶段地处理这个问题。当看到进位（两个 1 相加就会产生一个进位）结果表的时候，或许你已经看出来逻辑门和二进制加法的一些相关性了。

+进位	0	1
0	0	0
1	0	1

你可能意识到了，这和上一章中与门的输出结果是一样的。

AND	0	1
0	0	0
1	0	1

因此，利用与门可以计算两个二进制数加法的进位。

到此，我们着实取得了一些进展。下面我们要做的就是利用继电器来实现下表。

+加法	0	1
0	0	1
1	1	0

这是在作两个二进制数加法时需要解决的另一个问题。加法位的情况并不像进位位那样简单，但是我们即将实现它。

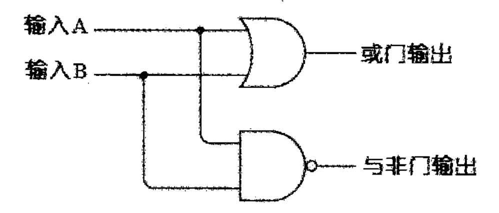
首先我们要知道，或门和我们想要的结果很相似，除了右下角的结果。

OR	0	1
0	0	1
1	1	1

与非门同样和我们想要的结果很相似，除了左上角的结果。

NAND	0	1
0	1	1
1	1	0

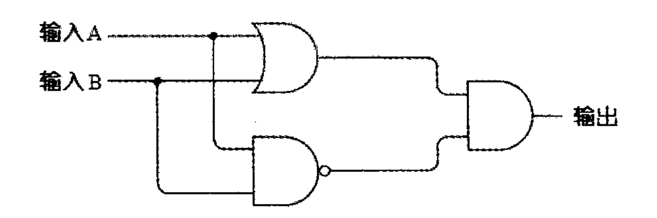
下面我们将或门和与非门连接到相同的输入上，如下图所示。



下表总结了或门和与非门的输出，并将其与我们想要的结果进行了对比。

输入 A	输入 B	或门输出	与非门输出	想要的结果
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

注意，我们想要的是 1，那么这种情况只有在或门和与非门的输出都为 1 时才会出现。这表明两个输出端可以通过一个与门连接到一起。

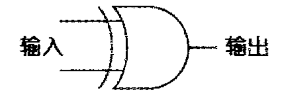


这就是我们想要的结果。

注意，在整个电路中仍然有两个输入和一个输出。两个输入同时作为或门和与非门的输入。或门和与非门的输出又分别作为一个与门的输入，最后得出了我们想要的结果。

输入 A	输入 B	或门输出	与非门输出	与门输出
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

实际上这个电路有个专门的名称，叫做异或门，简写为 XOR。之所以称为异或门是因为若想其输出结果为 1，要么仅让输入 A 为 1，要么仅让输入 B 为 1，两输入端都为 1 则输出为 0。为了不把或门、与非门和与门都画出来，我们可以使用一个电气工程师所采用的特定电气符号来表示异或门。



异或门在输入端比或门多出了一条曲线，除此之外它看上去和或门非常相像。异或门的特征如下表所示。

XOR	0	1
0	0	1
1	1	0

异或门是本书中详细介绍的最后一个逻辑门（第 6 个门有时会在电气工程中介绍到。它称做同或门，因为只有当两个输入相同的时候，其输出才为 1。同或门所给出的输出刚好与异或门相反，因此同或门的符号和异或门相同，但在输出端多了个小圆圈）。

让我们回顾一下到目前为止所了解的内容。将两个二进制数相加将产生一个加法位和一个进位位。

+加和	0	1
0	0	1
1	1	0

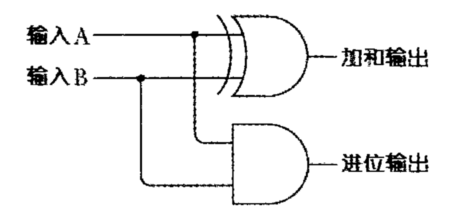
+进位	0	1
0	0	0
1	0	1

可以利用下面这两个逻辑门来实现这些结果。

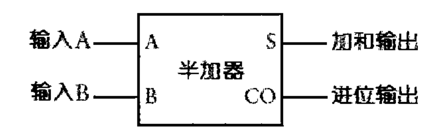
XOR	0	1
0	0	1
1	1	0

AND	0	1
0	0	0
1	0	1

两个二进制数相加的结果是由异或门的输出给出的，而进位位是由与门的输出给出的。因此我们可以将与门和异或门连在一起来计算两个二进制数（即 A 和 B）的和。



为了避免重复画与门和异或门，你可以采用如下这种简单的表示方式。

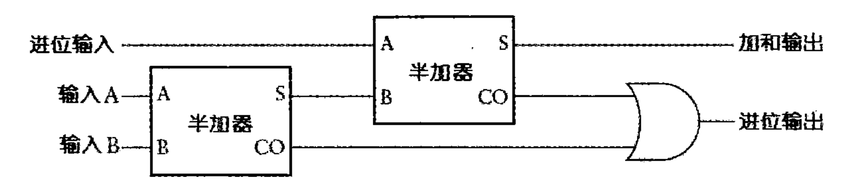


这个符号被称为半加器（Half Adder）。之所以叫半加器是有原因的，它将两个二进制数相加，得出一个加法位和一个进位位。但是绝大多数二进制数是多于 1 位的。半加器没有做到的是将之前一次的加法可能产生的进位位纳入下一次运算。例如，假设我们要将如下两个二进制数相加。

$$\begin{array}{r} 1111 \\ +\ 1111 \\ \hline 11110 \end{array}$$

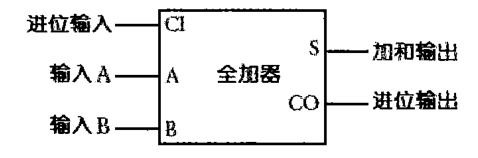
我们只能将半加器用于最右面一列的相加：1 加 1 等于 0，进位 1。对于从右面算起的第二列，由于进位位的存在，实际上需要将三个二进制数相加，而随后每一列的加法都是这样的。随后的每一列二进制数相加都需要将进位位算进来。

为了对三个二进制数进行加法运算，我们需要将两个半加器和一个或门做如下连接。



要理解它的工作原理，首先从最左边第一个半加器的输入 A 和输入 B 开始，其输出是一个加和及相应的进位。这个和必须与前一列的进位输入相加，然后再把它们输入到第二个半加器中。第二个半加器的输出和是最后的结果。两个半加器的进位输出又被输入到一个或门中。你可能会觉得，这里还需要一个半加法器，这当然是可行的。但是如果你了解了所有的可能性之后，你会发现，两个半加法器的进位输出是不会同时为 1 的。或门在这里已经足够，因为或门除了在输入都为 1 的时候以外，其他情况下结果和异或门结果相同。

为了避免重复地画上面的那个图，我们用以下形式来替代上图中的一堆符号，它称为全加器（Full Adder）。



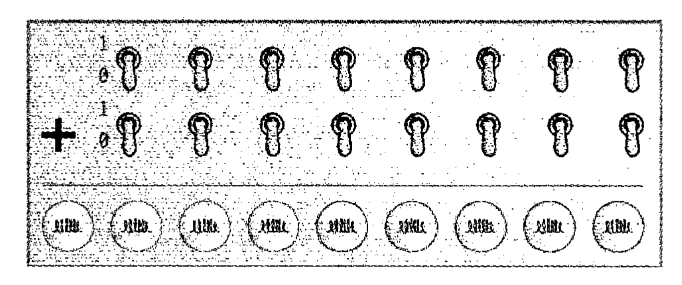
以下表格总结了全加法器所有可能的输入组合以及对应的输出结果。

输入 A	输入 B	进位输入	加和输出	进位输出
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

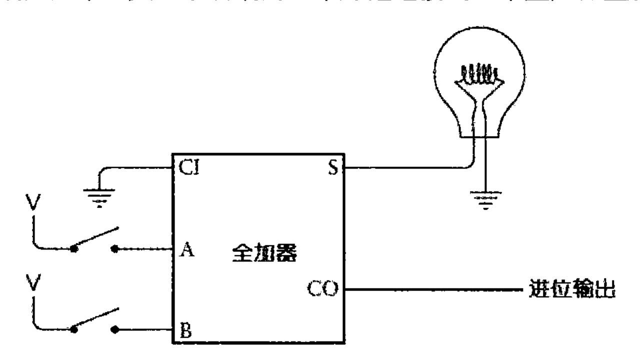
在本章前面曾经提到过，我们的加法器需要 144 个继电器。下面就来解释一下这个数目

是如何得到的。每个与门、或门和与非门都需要两个继电器，因此一个异或门中就包含 6 个继电器。一个半加器是由一个异或门和一个与门组成的，因此一个半加器就需要 8 个继电器。每个全加器由两个半加器和一个或门组成，所以它要 18 个继电器。我们需要 8 个全加器来制作 8 位二进制加法器。因而总共需要 144 个继电器。

再来看看之前提到的由灯泡和开关所组成的控制面板。现在我们可以开始将开关和灯泡连接到全加器了。

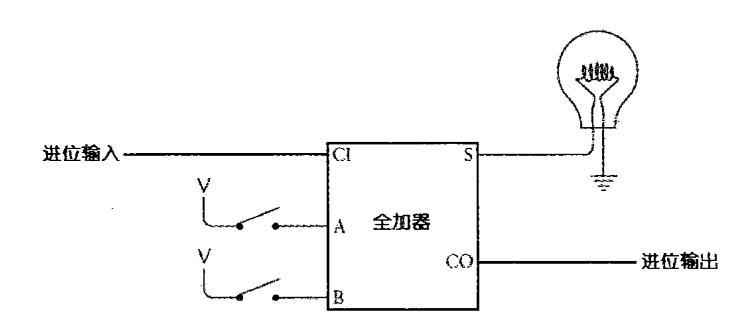


首先将最右端的两个开关和最右端的一个灯泡连接到一个全加器上。



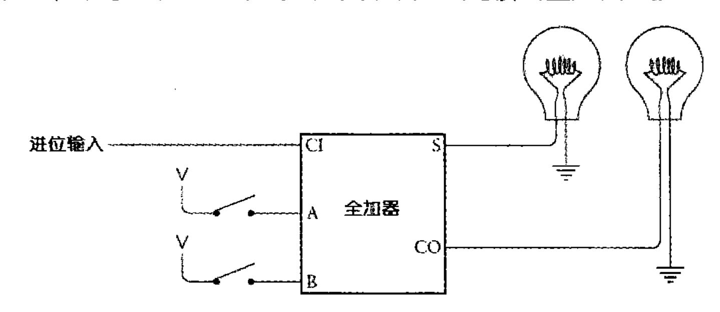
当把两个二进制数相加时，第 1 列的处理方式与其他列有所不同。因为后面的几列可能包括来自前面加法的进位，而第 1 列不会，所以全加器的进位输入端是接地的，这表示第 1 列的进位输入是一个 0。第 1 列二进制数相加后很可能会产生一个进位输出，这个进位输出是下一列加法的输入。

对于接下来的两个二进制位和灯泡，可以按如下办法来连接全加器。



第一个全加器的进位输出就是第二个全加器的进位输入。随后的每列二进制数都以同样的方式连接。每一列进位输出都是下一列的进位输入。

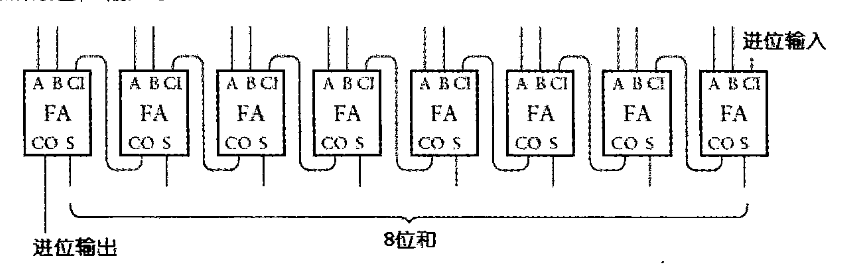
最终，第 8 个灯泡和最后一对开关将以如下方式连接到全加法器上。



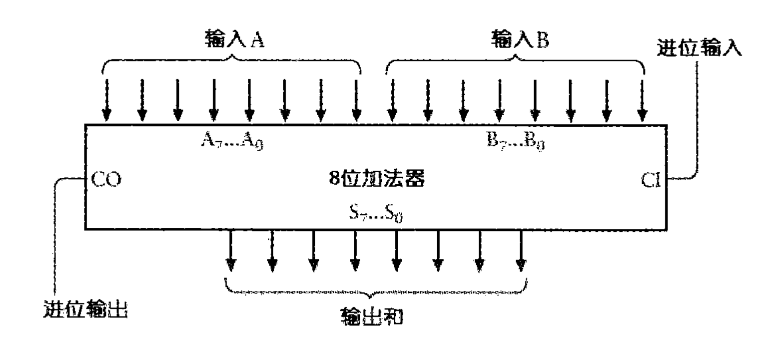
这里，最后一个进位输出将被连接到第 9 个灯泡上。

至此，我们就大功告成了。

还可以用另一种方式来看这 8 个全加器的连接，每个全加器的进位输出都作为下一个全加器的进位输入。



下面是画成一个盒子的完整的 8 位二进制加法器，输入标记为 $A_0 \sim A_7$ 和 $B_0 \sim B_7$，输出标记为 $S_0 \sim S_7$。



这就是表示多位数字中各位数字的常用方法。A_0、B_0和 S_0是最低有效位，或者说是最右边的一位。A_7、B_7和 S_7是最高有效位，或者说是最左边的一位。例如，下面演示了这样一列带有下标的字母是如何用来表示一个二进制数 0110-1001 的。

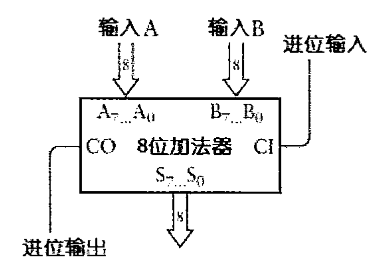
A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	1	1	0	1	0	0	1

下标从 0 开始，并且向着高有效位的方向递增，因为它们和 2 的乘方数（幂）相对应。

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	0	1	0	0	1

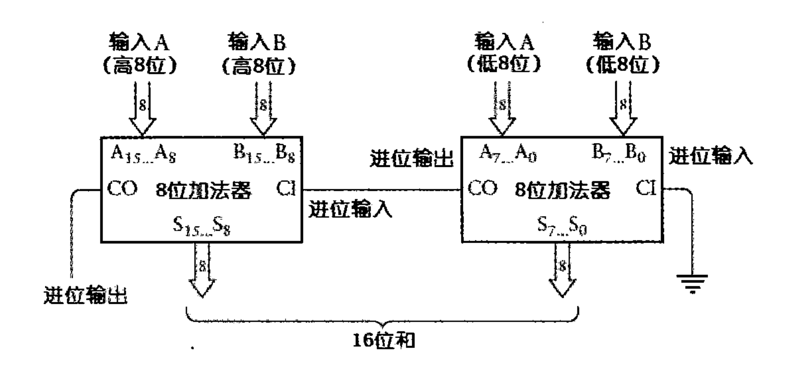
如果把下面一排的每个二进制位和其对应的 2 的幂相乘再依次相加，你就会得到 0110-1001 的十进制数表示 64 + 32 + 8 + 1，即 105。

另一种 8 位二进制加法器可用下图表示。



双线箭头包含了 8 个输入端，代表一组 8 个独立的信号。它们被标识为 $A_7...A_0$、$B_7...B_0$、$S_7...S_0$，同样也表示一个 8 位二进制数。

一旦你搭建起了 8 位二进制加法器，你就可以再搭建另外一个加法器。把它们级联起来就可以很容易地扩展出一个 16 位加法器。



右边加法器的进位输出被连接到了左边加法器的进位输入上。左边加法器的输入包含了两个加数的高 8 位，而得到的结果也是最终加和的高 8 位。

你可能会问："这真的就是计算机进行加法运算时所采用的方式么？"

基本上来说，是的。但也并不完全是。

首先，可以制作一个比这个算得更快的加法器。如果你看一下这个电路是如何工作的，最低有效位的一对数字相加所得出的一个进位输出，将要参与接下来的一对数字的加法运算，由此得到的一个进位输出又要参与再下一对数字的加法运算，依此类推。加法器的总体速度等于数字的位数乘以全加器器件的速度，这被称做行波进位（ripple carry，或脉冲进位）。更快的加法器运用了一个被称为"前置进位"的电路来提高运算的速度。

其次，也是最重要的，计算机已经不再使用继电器了！尽管它曾经被使用过。第一台数字计算机在 20 世纪 30 年代被建造完成，当时所使用的就是继电器，后来也使用过真空管。今天的计算机使用的是晶体管。在被用到计算机中时，晶体管的工作方式与继电器基本相同，但是正如我们即将了解到的，晶体管要比继电器计算速度更快、体积更小，而且噪声更弱、耗能也更低，而且更便宜。搭建一个 8 位加法器依然需要 144 个晶体管（如果你用前置进位法代替行波进位，将会用到更多的晶体管），但是电路却是极小的。

13

如何实现减法

当你确信继电器连接到一起真的可以实现二进制数加法的时候，你可能会问："那么如何实现减法呢？"本章后续的内容会帮你解答这个问题，因此提出这样的问题并不是说你在没事找事，而实际上这表明你是相当有察觉力的。加法和减法在某些方面相互补充，但在机制方面这两个运算则是不同的。加法是始终从两个加数的最右列向最左列进行计算的。每一列的进位加到下一列中。在减法中没有进位，而是有借位——一种与加法存在本质区别的麻烦机制。

例如，我们来看一个典型的借位减法的题目：

```
  253
- 176
-----
  ???
```

要解决这个问题，首先从最右列着手。我们看到，6 是大于 3 的，因此从 5 上借 1，再用 13 减去 6，得到结果为 7。由于我们已经在 5 上借了 1，因此，现在实际上那一位是 4，而 4 是小于 7 的，因此继续从 2 上借 1，14 减 7 结果为 7。而由于在 2 上借了 1，实际上这一位是 1，从中减去 1，结果为 0。因此，最后的结果应为 77：

$$
\begin{array}{r}
253 \\
-\ 176 \\
\hline
77
\end{array}
$$

如何才能通过一连串逻辑门来实现这个反逻辑呢？

然而，我们并不打算这样做。相反，我们打算用一个小技巧来让减法不涉及借位。这会使波洛尼厄斯[1]（既不是欠债人也不是出借人）满意，我们其他人也一样。此外，由于减法与计算机中以二进制编码的存储有关，详细地了解减法也是很重要的。

为了便于表达，将进行减法的两个数分别用被减数（minuend）和减数（subtrahend）表示。从被减数中减去减数，得到的结果叫做差（difference）。

$$
\begin{array}{r}
\text{被减数} \\
-\quad \text{减数} \\
\hline
\text{差}
\end{array}
$$

为了避免借位，首先要从 999 中减去减数，而不是从原来的被减数中减去减数。

$$
\begin{array}{r}
999 \\
-\ 176 \\
\hline
823
\end{array}
$$

由于操作数是三位数，所以这里使用 999。如果操作数是 4 位，则用 9999。从一串 9 中减去一个数叫做对 9 求补数。176 对 9 的补数是 823。反之亦然：823 对 9 的补数是 176。这样的好处就是无论减数是多少，计算对 9 的补数都不需要借位。

计算出减数对 9 的补数后，将补数与原来的被减数相加：

1 波洛尼厄斯是莎士比亚著名戏剧《哈姆莱特》中的一位世故的御前大臣，他在第一幕第三场向他即将离家外出的儿子说了一大段告诫的话，其中有一句：“不要向人告贷，也不要借钱给人。因为向人告贷的结果，容易养成因循懒惰的习惯；而把债款放了出去，往往不但丢了本钱，而且还失去了朋友。”后来，美国经济学家范里安在其所著的《微观经济学：现代观点》一书中使用了“波洛尼厄斯”一词，该词就成为了经济学中的一个概念。经济学在研究消费者进行跨时期的消费行为时，把这种既不向别人借钱、也不借钱给别人的状态叫作“波洛尼厄斯点”。因此，这里的“波洛尼厄斯”也就表示既不向别人借钱、也不借钱给别人的人。——译者注

$$\begin{array}{r} 253 \\ +\ 823 \\ \hline 1076 \end{array}$$

最后再将结果加 1，并减去 1000。

$$\begin{array}{r} 1076 \\ +\ 1 \\ -\ 1000 \\ \hline 77 \end{array}$$

到此，我们就得到了结果。答案与先前的相同，而且没有用到借位。

为什么这种方法行得通呢？原题目是这样的：

$$253 - 176$$

在这个式子中加上一个数再减去这个数，结果是相同的。因此先加上 1000，再减去 1000：

$$253 - 176 + 1000 - 1000$$

这个式子与下式等价：

$$253 - 176 + 999 + 1 - 1000$$

然后用以下方式将数字重新组合：

$$253 + (999 - 176) + 1 - 1000$$

这个式子与刚才描述过的用 9 的补数进行的计算是相同的。我们用两个减法和两个加法来替代一个减法，而在这个过程中避免了烦琐的借位。

如果减数大于被减数会怎么样呢？例如以下问题：

$$\begin{array}{r} 176 \\ -\ 253 \\ \hline ??? \end{array}$$

通常遇到这个问题时你可能会说："这里减数大于被减数，因此要将减数和被减数交换来执行减法，然后给结果取个相反数。"你可能在脑子里将这两个数交换，而写出这样

的答案：

$$\begin{array}{r} 176 \\ -\ 253 \\ \hline 77 \end{array}$$

如果希望求解这个问题而不使用借位的话，就要采用与之前稍微不同的方法。首先要像前面一样，用 999 减去减数 253，计算出对 9 的补数：

$$\begin{array}{r} 999 \\ -\ 253 \\ \hline 746 \end{array}$$

把该数对 9 的补数与被减数相加：

$$\begin{array}{r} 176 \\ +\ 746 \\ \hline 922 \end{array}$$

在前面的例子中，下一步应该加 1，并减去 1000 来得到最终结果。但是在这里，这种方法并不适用。因为你会遇到 923 减去 1000 的情况，这又导致了借位。

由于我们之前已经加了 999，这里再减去 999：

$$\begin{array}{r} 922 \\ -\ 999 \\ \hline ??? \end{array}$$

到这里，我们会意识到这个问题的结果是负数，因此需要将减数与被减数交换，用 999 减去 922。这里没有用到借位，结果与我们期望的相同：

$$\begin{array}{r} 922 \\ -\ 999 \\ \hline -\ 77 \end{array}$$

同样的技巧可以用于二进制数中，而且实际上这要比十进制数简单。让我们一起来看看该如何操作。

原来的减法题目是：

```
  253
- 176
  ???
```

将这些数字转化为二进制数，问题变为：

```
  11111101
- 10110000
  ????????
```

第一步，用 11111111（即 255）减去减数：

```
  11111111
- 10110000
  01001111
```

当计算十进制数减法的时候，减数是从一串 9 中减去的，结果称为 9 的补数。在二进制数减法中，减数是从一串 1 中减去的，结果称为 1 的补数。但是请注意，我们在求对 1 的补数时并不需要用到减法。在求对 1 的补数时，只需将原来的二进制数中的 1 变为 0，将 0 变为 1 即可。因此对 1 求补数有时也会称为相反数（negation）或反码（inverse）。这里你可能会想起第 11 章中的反向器，它的作用就是将 0 变为 1，将 1 变为 0。

第二步，将减数对 1 的补数与被减数相加：

```
   11111101
+  01001111
  101001100
```

第三步，将上式所得结果加 1：

```
  101001100
+         1
  101001101
```

第四步，减去 100000000（即 256）：

```
  101001101
- 100000000
    1001101
```

结果就等于十进制数的 77。

我们把这两个数颠倒位置后再做一遍。在十进制中，减法题目对应于：

```
  176
- 253
  ???
```

而用二进制表示为：

```
  10110000
- 11111101
 ?????????
```

第一步．用 11111111 减去减数，得到对 1 的补数：

```
  11111111
- 11111101
  00000010
```

第二步，将减数对 1 的补数与被减数相加：

```
  10110000
+ 00000010
  10110010
```

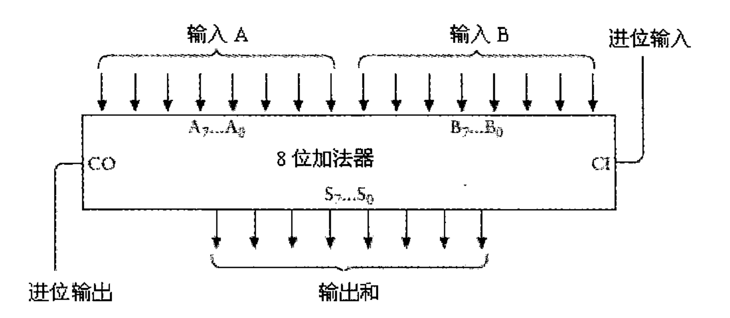
现在我们要用某种方法在结果中减去 11111111。当减数小于被减数的时候，我们将结果加 1 再减 100000000 来完成计算。但是你无法在不借位的情况下做到这一点。所以，我们先用 11111111 减去所得结果：

```
  11111111
- 10110010
  01001101
```

这里又一次用到了将各位取反来求得结果的方法，但是这个结果是 77，而真正的答案应该是–77。

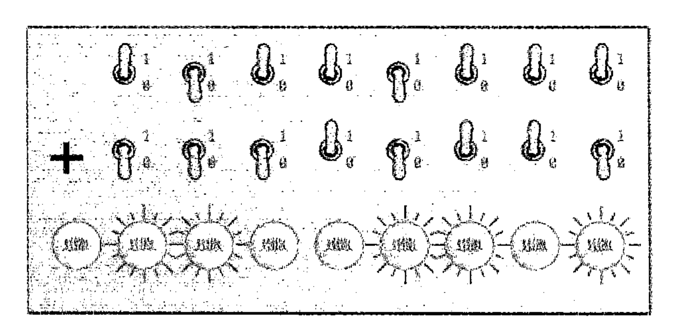
到这里，我们已经有了足够的条件来改造上一章所搭建的加法器，并让它像实现加法一样来实现减法运算。为了不让问题太复杂，这个新的加/减法器只执行在减数小于被减数的减法操作，即结果为正数的操作。

该加法器的核心是由逻辑门集成的 8 位全加器。



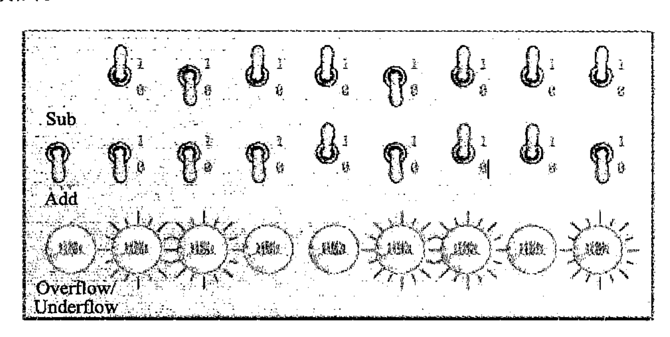
你可能还记得，输入 $A_0 \sim A_7$ 及 $B_0 \sim B_7$ 与两排分别表示两个要相加的 8 位二进制数的开关相连。进位输入接地。$S_0 \sim S_7$ 与表示结果的 8 个灯泡相连。由于这个加法有可能得到 9 位数值，进位输出端也连接了一个灯泡。

控制面板示意如下图所示。



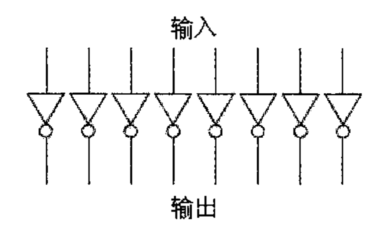
在上图中，开关所表示的是 183（即 10110111）与 22（即 00010110）相加，结果如灯泡所示为 205（即 11001101）。

8 位加/减法器所用的新面板较从前做了些许的改动。它增设了一个开关，用以选择做加法还是做减法。

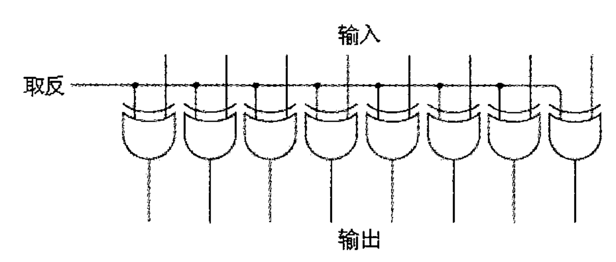


如上图所示，这个开关向下断开时表示选择加法运算，反之向上接通则表示选择减法运算。此外，右侧的 8 个灯泡用于表示计算结果。这里，第 9 个灯泡表示“上溢/下溢”。这个灯泡表明了正在计算的数字是一个不能用 8 个灯泡来表示的数字。如果在加法中得到了大于 255（上溢，overflow）或在减法中得到了负数（下溢，underflow）这个灯泡就会发光。当减数大于被减数的时候，就会得到一个负数。

加法器中新增的主要部分就是一个用来求 8 位二进制数对 1 补数的电路。之前提到，二进制数对 1 求补数相当于对其每位取反，因此我们计算 8 位二进制数补数的时候可以简单地应用 8 个反向器。



问题是，该电路只会对输入求反，而我们要的是一台既能做加法又能做减法的机器，因此就要求该电路当且仅当进行减法运算时才实现反转。电路可以改造为如下图所示。

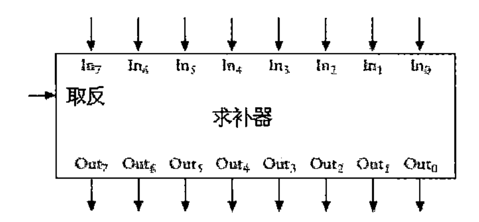


标记为“取反”的信号将被输入到每一个异或门中。回想一下异或门的工作方式，如下表所示。

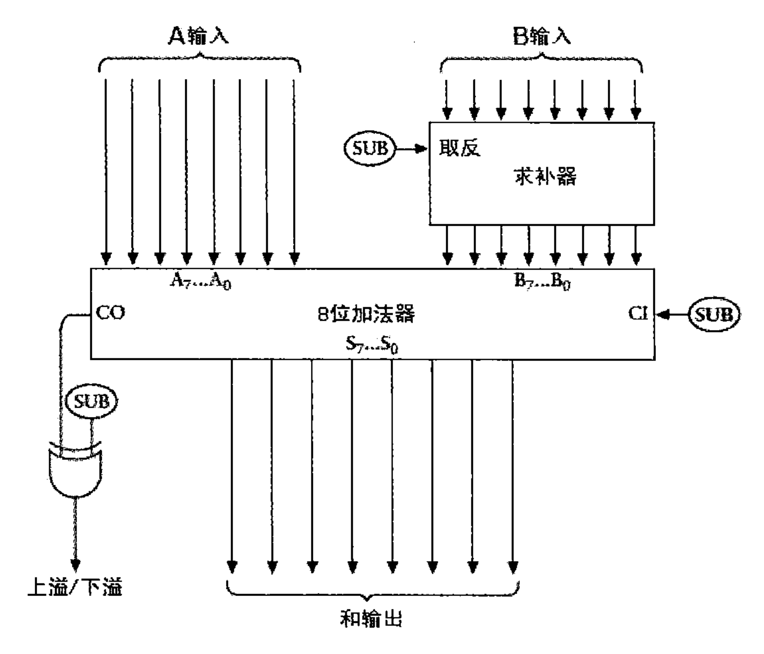
XOR	0	1
0	0	1
1	1	0

因此，如果“取反”信号是 0，则 8 个异或门输出与输入相同。例如，如果输入是 01100001，那么输出也为 01100001。如果“取反”信号为 1，则输出信号反置。例如，如果输入为 01100001，输出则为 10011110。

将 8 个异或门合并起来画成一个器件，称为求补器（One's Complement），如下所示。



将一个求补器，一个 8 位二进制加法器和一个异或门做如下连接。



注意，这里三个信号都标识为“SUB”，这就是加/减法转换开关。当该信号为 0 的时候，其进行的是加法运算，为 1 时进行的则是减法运算。在减法中，输入 B（第二排开关）在送入加法器之前，需先通过求补电路进行取反。此外，在做减法时，我们通过设定 CI（进位输入）为 1 来使得结果加 1。而在加法中，求补电路将不起作用，且输入 CI 为 0。

加法器的 SUB 信号和 CO（进位输出）输出作为异或门的输入来控制表示上溢/下溢的灯泡。如果 SUB 信号为 0（表示进行加法运算），则当加法器 CO 输出为 1 时灯亮，意思是加法计算结果大于 255。

当进行减法运算的时候，如果减数（输入 B）小于被减数（输入 A），这时加法器的

CO 输出为 1。这表示减法的最后一步要减去 100000000。也就是说减数要大于被减数，结果为负。上面所示器件现在还不能表示负数。因此，上溢/下溢指示灯仅在加法器的 CO 输出为 0 时才会亮起。

你应该会庆幸自己问了这个问题："减法该如何实现呢？"

本章一直在谈论负数，但是并没有提到负数在二进制中是如何表示的。你可能设想二进制会同十进制一样应用传统的负数符号。例如，–77 在二进制中为–1001101。这样做当然可以，但是应用二进制数的目的恰恰就在于只希望用 0 和 1 来表示所有的东西，当然也包括负号。

当然，你可以简单地用一个二进制位来表示负号，当这一位为 1 的时候就表示负数，为 0 则表示正数，尽管这样也是可行的，但还远远不够。还有另一种方法可以解决负数的表示问题，而且它还可以很轻松地让负数和正数相加。这种方法的缺点是必须提前算一下可能遇到的所有数字的位数。

让我们来想一想。通常用来表示正数和负数的方法，其好处就在于能表示所有的正数、负数。我们将 0 想象为这个无限延伸的序列的中点。这个序列中正数沿着一个方向延伸，而负数则按照另一个方向延伸：

…–1000000，–999999…–3，–2，–1，0，1，2，3…999999，1000000…

但是，如果我们并不需要无限大或无限小的数，而且在开始的时候我们就可以预知所使用的数字的范围，那情况就有所不同了。

以支票账户为例，这里人们通常会遇到负数。假设我的账户余额不超过 500 美元，并且银行给了我们 500 美元的无息预支额度，意思就是账户余额数值应该是一个在 499 美元到–500 美元之间的数。假设我们不会一次取出 500 美元，我们也不会支出多于 500 美元的金额，这里我们用到的只是美元，不涉及美分。

这些假设表明账户能处理的额度是介于–500 到 499 之间，总共 1000 个数。这个约束说明只用三位十进制数，且不用负号就可以表示所有需要的数字。我们并不需要用到从 500 到 999 之间的正数，因为我们所需要的数的最大值为 499。因此从 500 到 999 的三位数可以用来表示负数。具体情况如下：

用 500 表示–500

用 501 表示–499

用 502 表示–498

……

用 998 表示–2

用 999 表示–1

用 000 表示 0

用 001 表示 1

用 002 表示 2

……

用 497 表示 497

用 498 表示 498

用 499 表示 499

也就是说，以 5，6，7，8 或 9 开头的三位数实际上表示的都是负数，而不是把数字写成这样：

–500，–499，–498 … –4，–3，–2，–1，0，1，2，3，4 … 497，498，499

用这种表示法，我们可以将它们写成：

500，501，502 … 996，997，998，999，000，001，002，003，004 … 497，498，499

注意，这就形成了一个循环排序。最小的负数（500）看起来像是最大正数（499）的延续。而数字 999（实际上是–1）是比 0 小 1 的第一个负数。如果我们在 999 上加 1，通常会得到 1000。由于我们处理的是三位数，这个结果实际上就是 000。

这种标记方法称为 10 的补数（ten's complement）。为了将三位负数转化为 10 的补数，我们用 999 减去它再加 1。也就是说，对 10 求补数就是对 9 的补数再加 1。例如想要得到–255 对 10 的补数，用 999 减去 255 得到 744，然后再加 1，得到 745。

你可能听说过："减一个数就等于加一个负数。"你可能会回答："实际上还是减去了这个数。"然而，利用 10 的补数，我们将不会再用到减法。所有的步骤都用加法来进行。

假设你有一个余额为 143 美元的支票账户。你开了一张 78 美元的支票，也就意味着要将一个值为负的 78 美元加到 143 美元上。–78 对 10 的补数为 999–078+1，即 922。因此新余额为 143 美元+922 美元，相当于 65 美元（忽略溢出）。如果我们又开了一张 150 美元的支票，需要在余额上加上–150，–150 对 10 求补数为 850。因此先前的余额加上 850 等于 915，就是新的账户余额。而这个余额实际上是–85 美元。

这样的机制在二进制中被称为 2 的补数。以 8 位二进制数为例。范围为 00000000～11111111，对应十进制中的 0～255。但是如果你还想表示负数的话，则以 1 开头的每个 8 位数都表示一个负数，如下表所示。

二进制数	十进制数
10000000	–128
10000001	–127
10000010	–126
10000011	–125
…	…
11111101	–3
11111110	–2
11111111	–1
00000000	0
00000001	1
10000010	2
…	…
01111100	124
01111101	125
01111110	126
01111111	127

现在所表示的数的范围是–128～+127。最高有效位（最左位）作为符号位（sign bit）。符号位中，1 表示负数，0 表示正数。

为了计算 2 的补数，则首先要计算 1 的补数，然后再加 1。这等价于将每位取反再加 1。例如，十进制数 125 写为二进制为 01111101。为了表示–125 的对 2 的补数，首先将 01111101 的每位取反，得到 10000010，再加 1，得到 10000011。可以根据前面的表格核实一下结果。用同样的步骤，每位取反再加 1，可以将数值还原。

这个系统为我们提供了一种不用负号就能表示正、负数的方法。同样也让我们自由地将正数和负数用加法法则相加。例如，将与–127 和 124 等价的两个二进制数相加。利用上面的表格，可以简单地写为：

```
  10000001
+ 01111100
----------
  11111101
```

结果等于十进制的–3。

要注意的是，这里涉及了上溢和下溢情况，即结果大于 127 或小于–128。例如，将 125 与它自身相加：

```
  01111101
+ 01111101
----------
  11111010
```

由于最高位为 1，结果代表一个负数，相当于十进制的数–6。将–125 与它本身相加也会出现同样的情况：

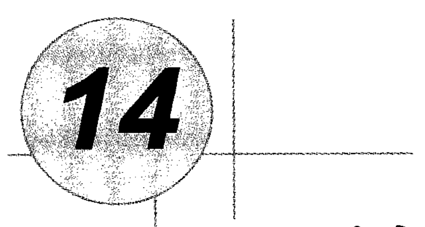
```
  10000011
+ 10000011
----------
 100000110
```

在一开始，我们规定所处理的数值为 8 位，因此最左位被忽略。右边 8 位相当于十进制的+6。

一般来说，如果两个操作数的符号相同，而结果的符号与操作数的符号不相同，这样的加法就是无效的。

现在，二进制数可以有两种不同的使用方法。二进制数可以是有符号的，也可以是无符号的。无符号的 8 位二进制数所表示的范围是 0～255。有符号的 8 位二进制表示的范围是–128～127。无论是有符号的还是无符号的，数字本身是无法显示的。例如，如果有一个人问："有一个 8 位二进制数，值为 10110110。它相当于十进制的多少？" 你必须先问："它是有符号数还是无符号数？它可能为–74 或者 182。"

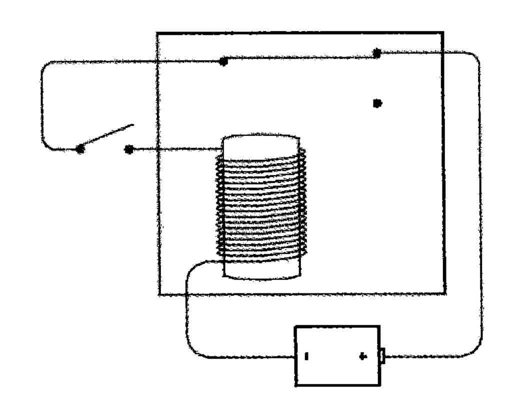
这就是二进制数的麻烦之处，它们只是一些 0 和 1，本身并没有任何含义。



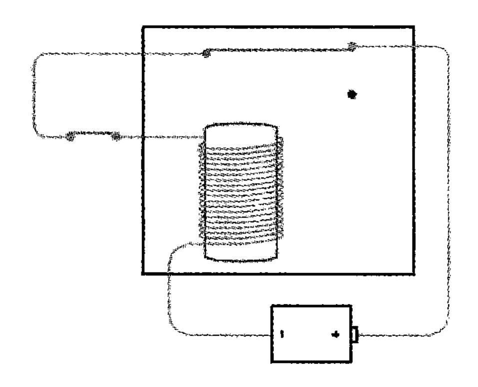
14 反馈与触发器

电可以让物体运动，这个道理人人都懂。只要稍微扫视一下我们的房间就会发现：很多电器中都装了电动机，比如钟表、电扇、食品加工器，以及 CD 播放器等。电同样可以使扩音器中的磁芯振动，正因为如此我们的音响设备、电视机才能够产生声音，播放语音和音乐。有一类设备或许能很清晰地阐释电能驱使物体运动的最简单也最具代表性的方式，然而由于这类设备正在被能够实现同样功能的电子器件逐步取代，它们正在迅速地消失。在我看来，最令人赞叹的例子应该算是电子蜂鸣器和电铃了。

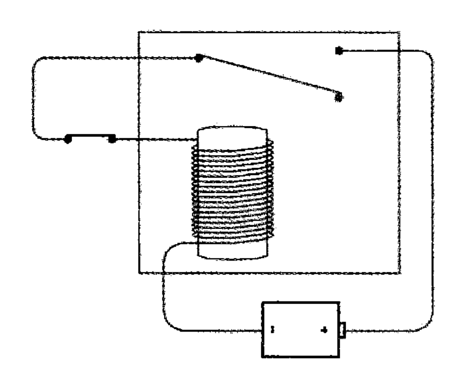
将继电器、电池、开关按如下形式连接。



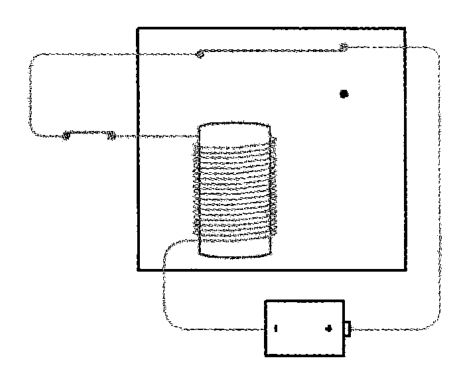
如果你认为这个系统看起来有点古怪，说明你还没有发挥出想象力。或许以前我们没见过采用这种连接方式的继电器，因为我们通常所见过的继电器，其输入和输出是分开的，而这里却构成了一个回路。当开关闭合后，电路就连通了。



连通的电路使得电磁铁把金属簧片拉了下来。

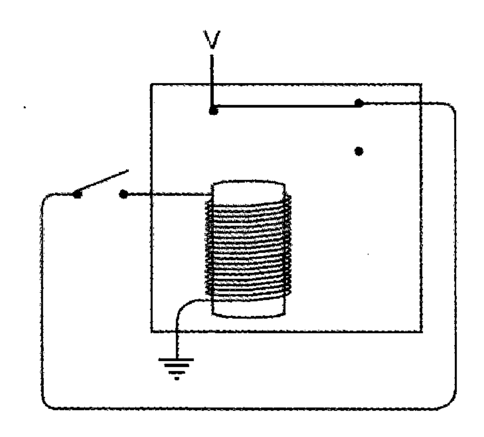


当金属簧片的位置变化时，电路不再连通，电磁铁不再具有磁性，金属簧片又弹回原位。

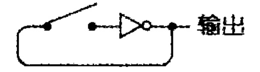


如此一来，电路又一次连通了。从整个过程来看：开关一旦闭合，金属簧片就会上下跳动——电路也会随之连通或断开——声音也就会随之发出。如果金属簧片发出了一种刺耳的声音，这套系统就成为了一个蜂鸣器。如果金属簧片前端是一把小锤子，旁边只要放上一个锣，就构成了一个电铃。

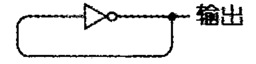
有两种方法可以使继电器连接成为一个蜂鸣器，下面再给出一种方法的描述，在示意图中包含了一个标准电压符号和一个接地符号。



看到这幅图你或许立刻想起了第 11 章介绍过的反向器，因此电路可以简化为如下图所示。



正如你所记得的那样，当反向器的输入是 0 的时候，它的输出就为 1；而当其输入为 1 时，输出就为 0。电路中的开关一旦闭合，反向器中的继电器就会在连通与断开这两种状态之间反复交替。你也可以将电路中的开关省去，这样就可以使反向器连续地工作，如下所示。



这幅图似乎在表达着一种矛盾的逻辑，反向器的输出与其输入是相反的，但是在这里，输出同时又是输入！然而，我们要牢牢记住，反向器在本质上就是一个继电器，而继电器将状态取反以得到另一个状态是需要一点点时间的。所以，即使输入和输出是相同的，输出也会很快地改变，成为输入的相反状态（当然，输出随即也会很快改变输入，

如此反复)。

电路的输出是什么呢？其实就是要么提供电压，要么不提供电压，在两者之间切换。我们也可以换种方式来表达——输出结果要么是 0，要么是 1。

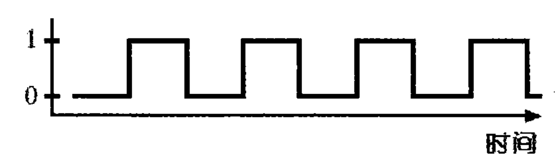
我们把这种电路称为振荡器（oscillator），它和我们先前学到的所有东西存在本质上的区别。在此之前我们讲过的所有的电路，其状态的改变都依靠人为的干预，通常是通过改变开关状态来实现的。但是振荡器却在不需要人干涉的情况下，可以完全自发地工作。

当然，单独的一个振荡器用处并不大，但是在本章的后面和接下来的几章里，我们会发现，在与其他电路连接后所组成的自动控制系统中，振荡器有着举足轻重的作用。为了使不同组件同步工作，所有计算机都配备着某种振荡器。

当采用 0 和 1 的交替序列来表示振荡器的输出时，我们一般使用下面这样的图来形象地描述输出。



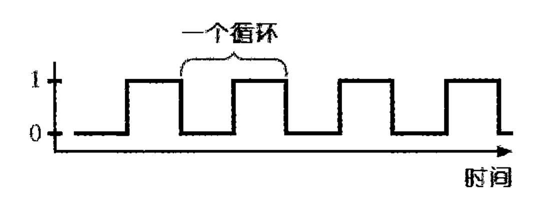
我们可以通过这幅图来充分地了解电路的输出，水平坐标代表时间，垂直坐标用来表示输出是 0 还是 1。



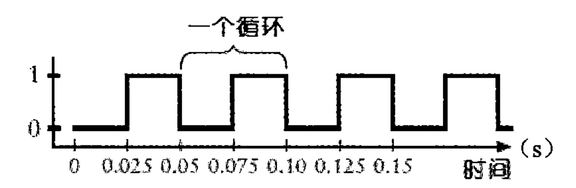
这幅图表示随着时间的推移，振荡器的输出在 0 和 1 之间按照固有的规律交替变化。正因为这一点，振荡器又经常被称为时钟（clock），通过振荡进行计数也是一种计时方式。

振荡器运行速度究竟有多快呢？换句话说，金属簧片多久会振动一次？或者每秒钟振动多少次呢？这很大程度上依赖于继电器的内部构造。你容易想到，一个又大又重的继电器只能缓慢地上下摆动；而一个又小又轻的继电器却可以高速地跳动。

振荡器从某个初始状态开始，经过一段时间又回到先前初始状态的这一段间隔定义为振荡器的一个循环（cycle），或者称为一个周期，如下图所示。



一个循环所占用的时间就是该振荡器的周期（period）。假设我们使用的振荡器的周期恰好是 0.05s，任取一个时间点，将其设置为起始状态点，我们把它标注为零点，就可以在水平轴上标出相应的时间。



周期的倒数就是振荡器的频率（frequency）。在这个例子中振荡器的周期是 0.05s，那么其频率就是 1 ÷ 0.05s，即振荡器每秒钟产生 20 次循环，而相应的输出每秒钟也变化 20 次。

每秒钟的循环次数与每小时穿越的英里数、每平方英尺的重量、每份食物的卡路里数等概念一样都是很容易理解的，但这种描述方法已不常用。为了纪念发送和接收无线电波的第一人——亨利希·鲁道夫·赫兹（1857-1894），后人使用“赫兹”这个词来表示这一概念。这种用法起源于 20 世纪 20 年代的德国，几十年之后逐渐被其他国家所广泛采纳。

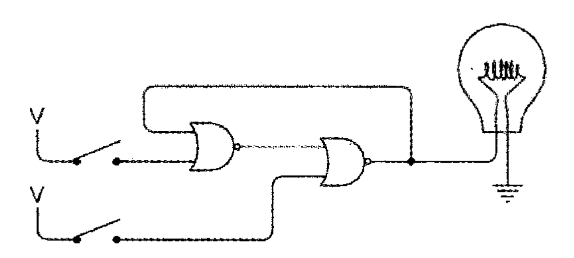
这样，上述振荡器的频率就是 20 赫兹，记做 20 Hz。

目前为止，我们还只是在猜测一个振荡器的速度。在本章后面我们将构建一种可以测量振荡器速度的元件。

在此之前，让我们先来看看采用特殊方式连接的一对或非门。或非门的特点是只有在两个输入端都没有电压时，输出端才产生电压。

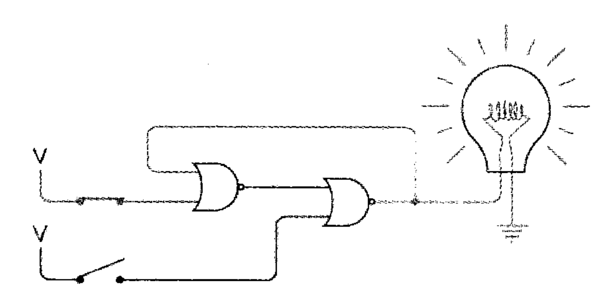
NOR	0	1
0	1	0
1	0	0

下面是一个包含两个或非门、两个开关和一个灯泡的电路。

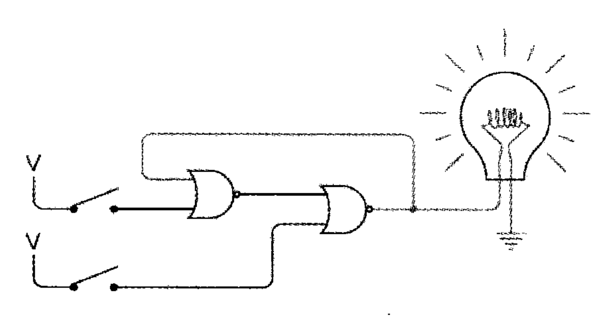


值得注意的是这种特殊的弯曲的连线方式：左边或非门的输出是右边或非门的输入，而右边或非门的输出是左边或非门的输入。这种连接方式我们称之为反馈（feedback）。系统的输出返回给输入这种形式和我们在振荡器中讨论的情况很相似。接下来你将会看到，本章大部分电路都具备这种特质。

在初始状态下，电路中只有左边的或非门输出电流，这是因为其两个输入均为 0。让我们闭合上面的开关，左边或非门将立刻输出 0，右边或非门的输出也会随之变为 1，这时灯泡将被点亮。

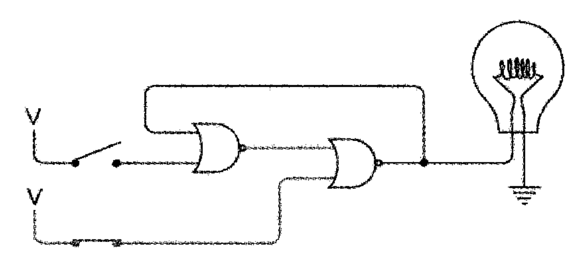


奇妙的是，这时一旦你关闭上边的开关，灯炮依然闪闪发光。这是因为由于左边或非门的输入中有一个为 1，其输出依然是 0，因而左边或非门的输出不变，所以灯泡仍然亮着。

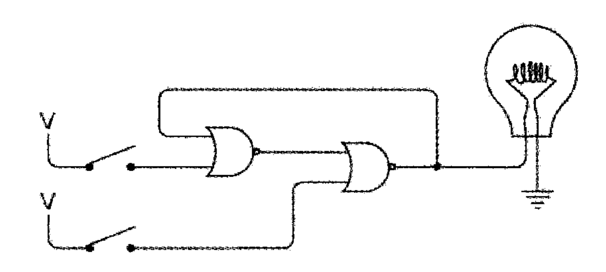


难道你不觉得有点奇怪吗？两个开关都断开——和第一幅场景描述的是一模一样——但这里的灯泡却仍发光。这与我们先前所见过的所有情况都完全不同。先前遇到的电路其输出依赖且仅依赖于其输入，这次的结论与以前的大相径庭。无论上面的开关怎么调整其状态，灯泡总是亮着。这个开关对电路毫无影响，究其原因可以发现这是由于左边或非门的输出一直为 0。

现在来试试闭合下面的开关。我们会发现右边或非门的输入中有一个立刻变为 1，其输出就相应地变为 0，灯泡随之熄灭。左边或非门的输出此刻变为 1。



这时你再去断开下面的开关就会发现，灯泡一直处在熄灭状态。



此时的电路状态与初始时是一样的。但是这次无论你怎么改变下面开关的状态，灯泡丝毫不受影响。我们将先前的情况一起总结一下：

- 接通上面的开关，灯泡被点亮，断开此开关灯泡仍然亮着。
- 接通下面的开关，灯泡被熄灭，断开此开关灯泡仍然不亮。

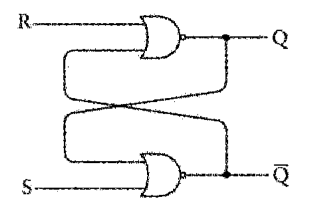
电路的奇怪之处是：同样是在两个开关都断开的状态下，灯泡有时亮着，有时却不亮。当两个开关都断开时，电路有两个稳定态，这类电路统称为触发器（Flip-Flop），Flip-Flop 这个单词也可以有“沙滩鞋”或者是“政治策略”的意思。触发器是在 1918 年被发明的，发明者是英国无线电物理学家威廉姆・亨利・艾克里斯（1875-1966）和 F.W. 乔丹（信息不详）。

触发器电路可以保持信息，它可以“记住”某些信息。特别地，对于本章先前所讲述的触发器，它可以记住最近一次是哪个开关先闭合。如果你遇到这样一种触发器，如果它的灯泡是亮着的，你就可以推测出最后一次连通的是上面的开关；而如果灯泡不亮则可推测出最后一次连通的是下面的开关。

触发器和跷跷板有着很强的相似性。跷跷板也有两个稳定状态，它不会长期停留在不稳定的中间位置。通过观察跷跷板，我们很容易推测出哪边最后一次被压下来。

尽管你现在可能还没感受到这一点，但触发器的的确确是一种必不可少的工具。它们可以让电路“记住”之前发生了什么事情。想象一下，如果你没有了记忆力，该如何去数数，我们不记得刚刚数过的数，当然也就无法确定下一个数是什么！同理，一个能计数的电路（本章后面要讲到）必定需要触发器。

触发器种类繁多，先前所讲述的是最简单的一种 R-S（Reset-Set，复位/置位）触发器。我们通常把两个非或门绘制成另一种形式，加上标识符就得到了下面这幅图。



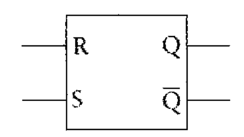
我们通常用 Q 来表示用于点亮灯泡的输出的状态。另一个输出 $\overline{Q}$（读做 Q 反）是对 Q 的取反。Q 是 0，$\overline{Q}$ 就是 1，反之亦然。输入端 S（Set）用来置位，R（Reset）用来复位。你可以把“置位”理解为把 Q 设为 1，而“复位”是把 Q 设为 0。当状态 S 为 1 时（对应于先前触发器中上面的开关闭合的情况），此时 Q 变为 1 而 $\overline{Q}$ 变为 0；当 R 状态为 1 时（对应于前面图中闭合下面的开关的情况），此时 Q 变为 0 而 $\overline{Q}$ 变为 1。当 S 和 R 均为 0 时，输出保持 Q 原来的状态不变。我们把结论总结如下表所示。

输入		输出	
S	R	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0	Q	$\overline{Q}$
1	1	禁止	

这类表称为功能表（function table）、逻辑表（logic table）或真值表（truth table）。它表达了不同输入组合所对应的不同输出结果。因为 R-S 触发器仅有两个输入端，所以不同的输入组合共有 4 种，分别对应于表中的 4 行。

注意表中倒数第 2 行；这一行输入 S 和 R 均为 0，而输出标识为 Q 和 $\overline{Q}$。这表示当 S 和 R 输入均为 0 时，Q 和 $\overline{Q}$ 端的输出保持为 S、R 同时被设为 0 以前的输出值。表中最后一行表示 S 和 R 均为 1 的输入组合是被禁止或者不合法的。不要误解为你会因此被逮捕，而是说如果 S、R 状态同时为 1 时，Q 和 $\overline{Q}$ 均会为零，这与 Q 和 $\overline{Q}$ 互反的假设关系相矛盾。所以当使用 R-S 触发器进行电路设计时，R、S 输入同时为 1 的情况一定要避免。

R-S 触发器可以简化为带有输入和输出标志的小框图，就像下面画的这样。



R-S 触发器最突出的特点在于，它可以记住哪个输入端的最终状态为 1。但是有时候我们需要一种记忆能力更加强大的电路，例如能记住在某个特定时间点上的一个信号是 0 还是 1。

在构造具备这种功能的电路之前，让我们先来思考一下它的具体行为。这个电路存在两个输入。其中一个我们称之为数据端（Data）。与所有数字信号一样，数据端取值为 0 或 1；另一个输入被称为保持位（Hold That Bit），保持位的作用就是使当前的状态被“记住”，通常情况下保持位被设置为 0，在这种情况下数据端对电路不产生影响。当保持位置 1 时，数据端的值就会在电路系统中被“记住”。随后保持位又置为 0，这时电路已经“记住”了数据端的最后一次输入，而之后数据端的输入无论如何变化都不会对电路产生影响。

我们可以把状态转化的过程以真值表的形式表示如下。

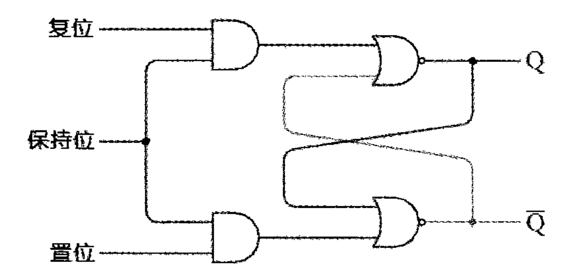
输入		输出
数据	保持位	Q
0	1	0
1	1	1
0	0	Q
1	0	Q

在前两种情况下，保持位为 1，输出 Q 与数据端输入相同；后面两种情况下，保持位为 0，输出端 Q 和其前一个状态保持一致。值得注意的是，保持位为 0 意味着输出将不再变化，也就是说不再被数据端所影响，我们可以进一步将真值表简化为如下所示。

输入		输出
数据	保持位	Q
0	1	0
1	1	1
X	0	Q

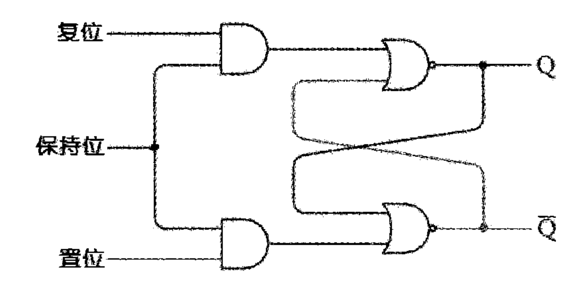
X 表示“其取值情况与结果无关”，只要保持位的值为 0，那么数据位对电路的输出没有影响，电路的输出和其前一个状态相同。

如果使用先前学过的 R-S 触发器来实现这种具有保持位的功能系统，那么我们的电路需要在输入端增加两个与门，下图所给出了该系统的实现电路。

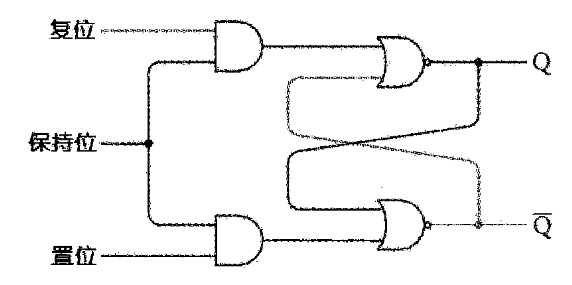


回忆一下与门，它的特点是只有在输入端都为 1 的状态下，输出才为 1。在上面这幅图中，输出端 Q 为 0，$\overline{Q}$ 为 1。

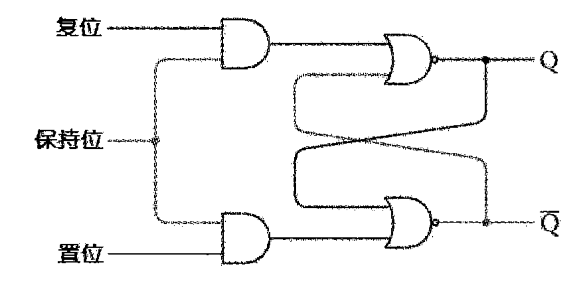
只要保持位为 0，则置位端对于输出结果不会有任何影响。



同样，复位信号对输出也无任何影响。



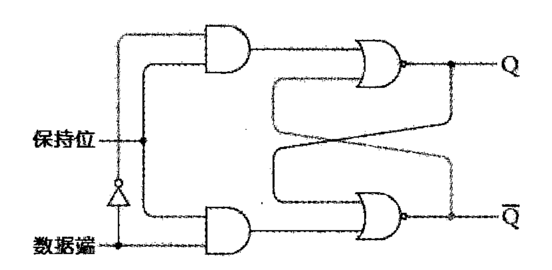
当保持位信号为 1 时，这套电路系统就和先前讲过的 R-S 触发器功能一致。



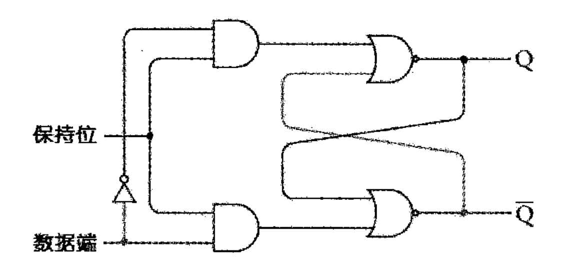
这时由于上面与门的输出和复位端输入相同，而下面与门的输出和置位端输入相同，所以电路系统的功能和普通的 R-S 触发器是一样的。

但是我们离目标还差一点。我们只想要两个输入，而不是三个，怎么解决这个问题呢？先回忆一下 R-S 触发器的功能表：两个输入端同时为 1 是非法的，要尽量避免；而两个输入端同时为 0 是无意义的，因为那种情况下输出就会保持不变。我们只要将保持位设置为 0，就完全可以实现相同的功能。

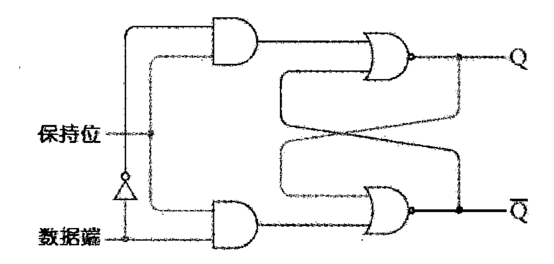
由此可以总结出，真正有意义的输入可以是 S 为 0，R 为 1 或者是 R 为 0，S 为 1 的情形。如果把数据端信号看做置位信号，把它取反后的值看做复位端信号，我们可以画出相应的电路图如下所示。



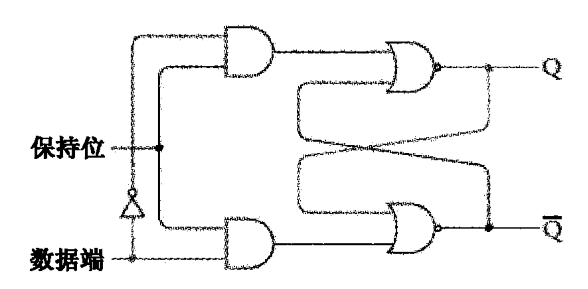
在上图所表示的情况下，所有输入均为 0，而输出 Q 也为 0（此时 $\overline{Q}$ 为 1）。可以看出只要保持位为 0，电路输出就丝毫不受输入端的影响。



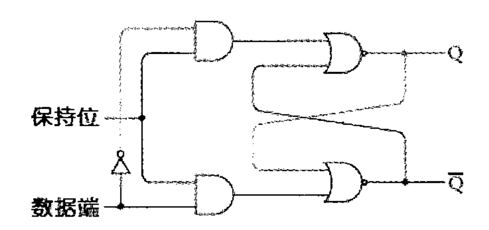
当保持位为 1 时，电路反映出数据端输入的值。



现在 Q 端的输出和数据输入是一致的，而 $\overline{Q}$ 端则正好相反。现在保持位又回到 0，如下图所示。

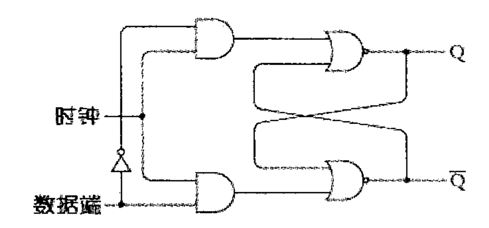


这时，电路会“记得”当保持位最后一次置 1 时数据端输入的值，数据端的变化对此没有影响。例如，数据端再置回 0 对输出将不会产生影响。



这个电路称为电平触发的 D 型触发器，D（Data）表示数据端输入。所谓电平触发是指当保持位输入为某一特定电平（本例中为“1”）时，触发器才保存数据端的输入值（很快，我们将看到另一种形式的触发器）。

通常情况下，当这种电路出现在书中的时候，输入端是不会被标记为保持位的，而是被标记为时钟（clock）。当然，这种信号并不是真正的时钟，但是在某些情况下它却具有类似时钟的属性，即它可以在 0 和 1 之间有规律地来回变化。但是现在时钟仅仅用来指示什么时候保存数据。



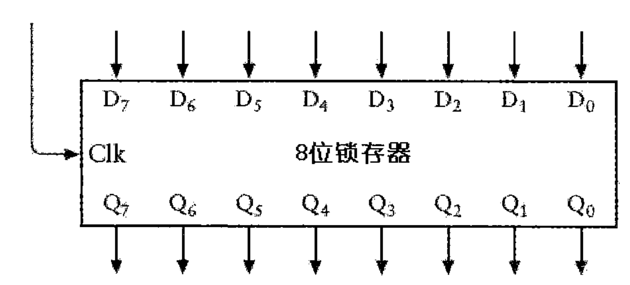
通常把数据端简写为 D，时钟端简写为 Clk，其功能表如下所示。

输入		输出	
D	Clk	Q	$\overline{Q}$
0	1	0	1
1	1	1	0
X	0	Q	$\overline{Q}$

这个电路也就是所谓的电平触发的 D 型锁存器，它表示电路锁存住一位数据并保持它，以便将来使用。这个电路也可以被称为 1 位存储器。在本书的第 16 章将会介绍如何将多个 1 位存储器连接起来构成多位存储器。

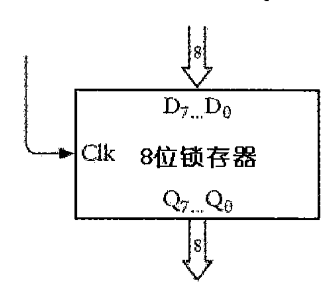
在锁存器中保存多位值通常是很有用的。假如你想用第 12 章的加法器把 3 个 8 位数相加，可以在开关的第 1 行中存入第 1 个加数，以同样的方式把第 2 个加数存入第 2 行，但是必须记下第一次相加的结果。然后你需要把这个结果输入到开关的一行中，再把第 3 个加数输入到开关的另一行中。而你实际上不必输入中间结果，你应该能够在第一次计算之后直接使用它。

可以使用锁存器来解决这个问题。我们在一个小盒子里布置 8 个锁存器，如前所述，每个锁存器包括两个或非门、两个与门以及一个反相器。所有的时钟输入端都互相连在一起。结果如下图所示。

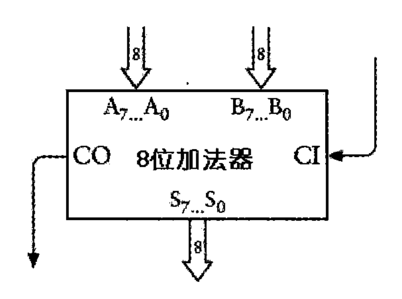


这个锁存器可以一次保存 8 位数。上面的 8 个输入端依次标记为 $D_0 \sim D_7$，下面的 8 个输出端被标记为 $Q_0 \sim Q_7$。左边的输入是时钟（Clk），时钟信号通常为 0。当时钟信号为 1 时，D 端输入的 8 位值被送到 Q 端输出。当时钟信号为 0 时，这 8 位值将保持不变，直到时钟信号再次被置 1。

也可以将 8 位锁存器的 8 个数据输入端和 8 个 Q 输出端画为两组线，如下图所示。

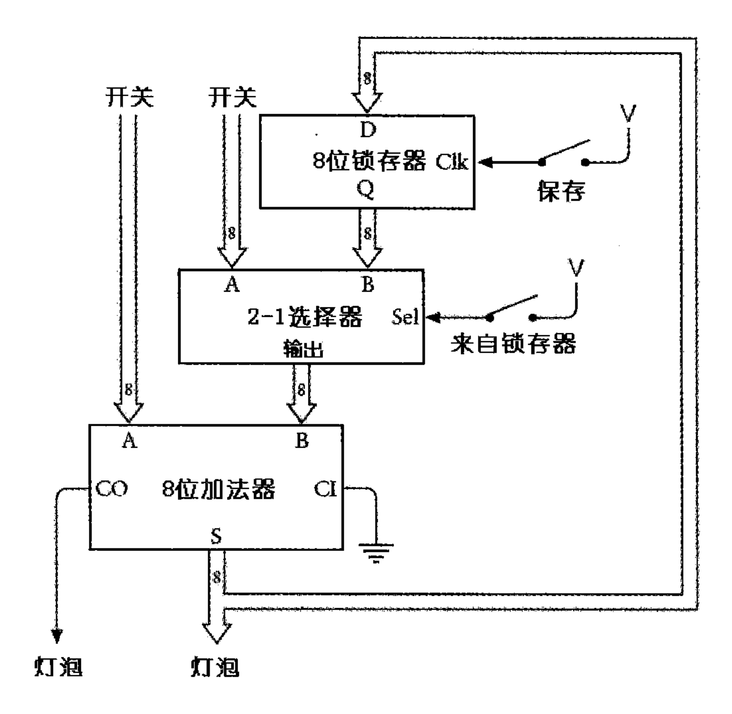


下面是 8 位加法器的图示。

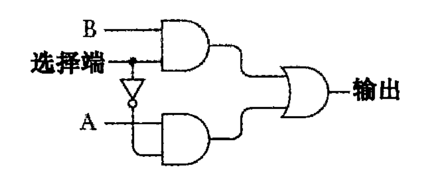


通常（先不考虑上一章讲到的减法器），8 个 A 输入端和 8 个 B 输入端连接到开关上，CI（进位输入）接地，而 8 个 S（计算和）输出以及 CO（进位输出）端连接到灯泡上。

经过改进，8 位加法器的 8 个 S 输出端既与灯泡相连，又连接到 8 位锁存器的数据（D）输入端。标记为“保存”（Save）的开关是锁存器的时钟输入，用来存放加法器的运算结果。



标识为 2-1 选择器的方块是让你用一个开关来选择加法器的 B 端输入是取自第 2 排开关还是取自锁存器的 Q 端输出。当开关闭合时，就选择了用 8 位锁存器的输出作为 B 端输入。2-1 选择器使用了 8 个如下所示的电路。



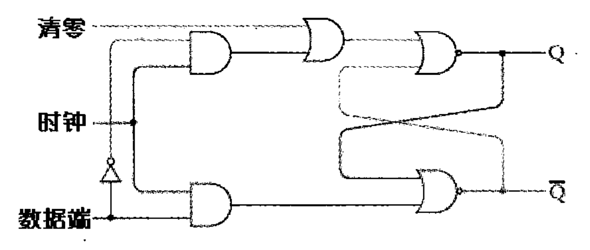
如果选择端（Select）输入是 1，那么或门的输出和 B 端的输入就是一致的。这是因为上面与门的输出和 B 端输入是一样的，而下面与门的输出是 0。类似的，如果选择端的输入是 0，那么或门的输出则和 A 端输入一致。总结起来如下表所示。

输入			输出
选择端	A	B	Q
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

改进后的加法器中包含了 8 个这样的 1 位选择器。所有的选择端输入信号都是连在一起的。

改进后的加法器不能很好地处理进位输出（CO）信号。如果两个数的相加使得进位输出信号为 1，那么当下个数被加进来的时候，这个信号将被忽略掉。一个可能的解决方案是将加法器、锁存器、选择器均设置为 16 位宽，或者至少应该比你可能遇到的最大的和的位数多一位。这个问题留到第 17 章具体讲述。

对于加法器来说，一个更好的改进方法是去掉一整排 8 个开关。但是首先要对 D 触发器做一些修改，为它加一个或门和一个称为清零（Clear）的输入信号。清零信号通常为 0，但当它为 1 时，Q 输出为 0，如下图所示。

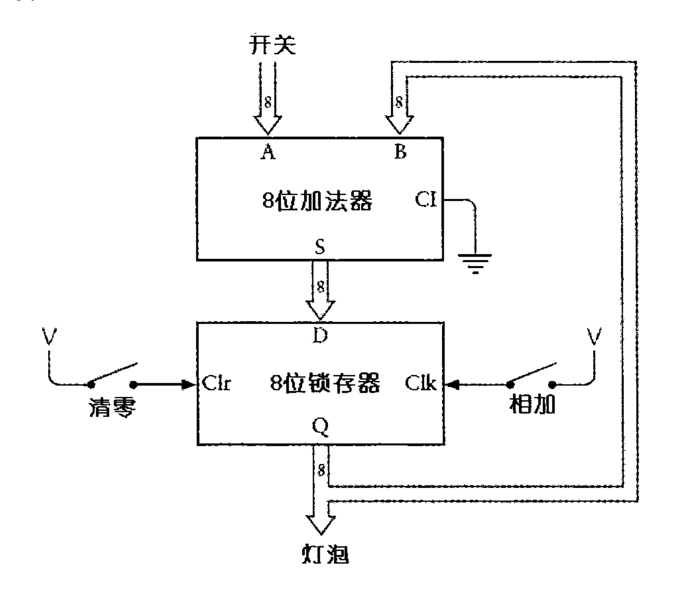


无论其他信号是什么，清零信号总是强制使 Q 输出为 0，以达到使触发器清零的目的。

也许你还不明白为什么要设置这个信号，为什么不能通过把数据输入端置 0 和把时钟输入端置 1 来使触发器清零呢？这也许是因为我们无法精确控制数据端的输入信息的缘故。我们可能有一组 8 个锁存器，它们连着 8 位加法器的输出端，如下图所示。

注意，标识为“相加”（Add）的开关现在控制着锁存器的时钟输入。

你可能会发现这个加法器比前面的那个好用，特别是当你需要加上一长串数字时。首先按下清零开关，这个操作会使锁存器的输出为 0，并且熄灭了所有的灯泡，同时使 8 位加法器的第 2 行输入全为 0。然后，通过开关输入第一个加数，并且闭合“相加”开关，这个加数的值就反映在灯泡上。再输入第二个加数并再次闭合“相加”开关。由开关输入的 8 位操作数加到前面的结果上，所得的和体现到灯泡上。反复如此操作，可以连续进行很多次加运算。

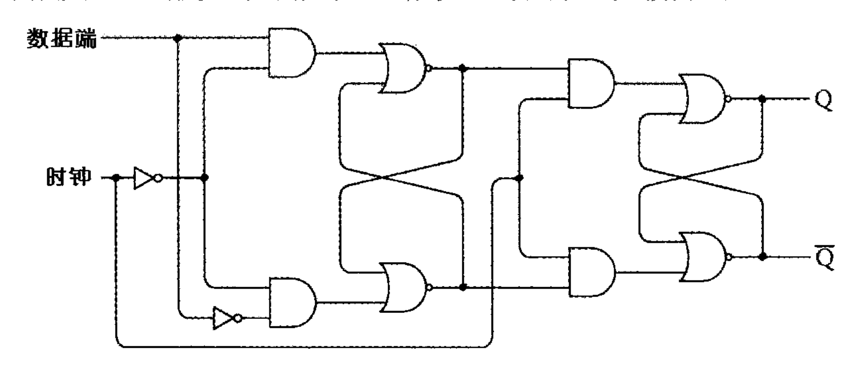


前面提到过，我们所设计的 D 触发器是电平触发的，也就是说为了使数据端的值保存在锁存器中，必须把时钟端的输入从 0 变为 1（即高电平）。但是，当时钟端输入为 1 时，数据端的输入是可以改变的，这时数据端输入的任何改变都会反映在 Q 和 $\overline{Q}$ 的输出值中。

对某些应用而言，电平触发时钟输入已经足够用了；但是对另外一些应用来说，边沿触发（edge-triggered）时钟输入则更有效。对于边沿触发器而言，只有当时钟从 0 跳变

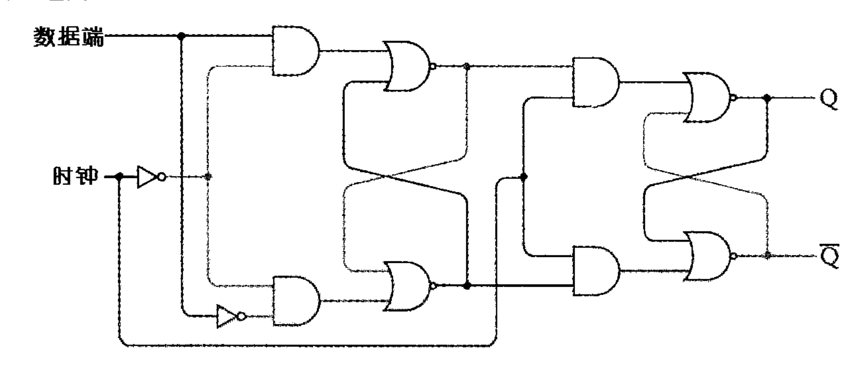
到 1 时，才会引起输出的改变。它们的区别在于，在电平触发器中，当时钟输入为 0 时，数据端输入的任何改变都不会影响输出；而在边沿触发器中，当时钟输入为 1 时，数据端输入的改变也不会影响输出。只有在时钟输入从 0 变到 1 的瞬间，数据端的输入才会影响边沿触发器的输出。

边沿触发的 D 型触发器是由两级 R-S 触发器按如下方式连接而成的。

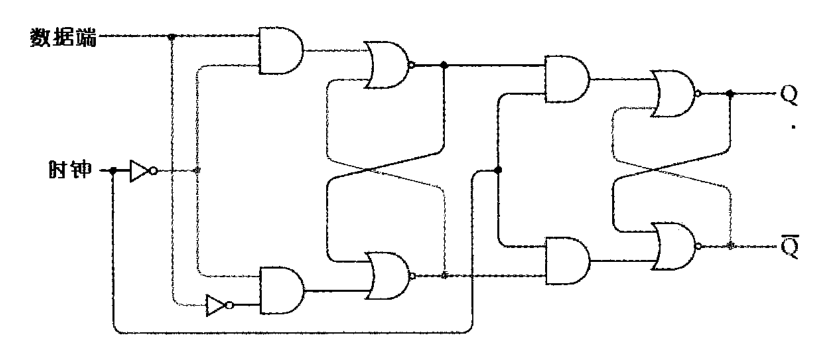


这里，时钟端的输入既控制着第一级 R-S 触发器，也控制着第二级，但是要注意的是时钟信号在第一级中进行了取反操作，这意味着除了当时钟信号为 0 时保存数据外，第一级 R-S 触发器和 D 型触发器工作原理完全一致。第二级 R-S 触发器的输出是第一级的输入，当时钟信号为 1 时，它们都被保存。一言概之，只有当时钟信号由 0 变为 1 时，数据端输入才被保存下来。

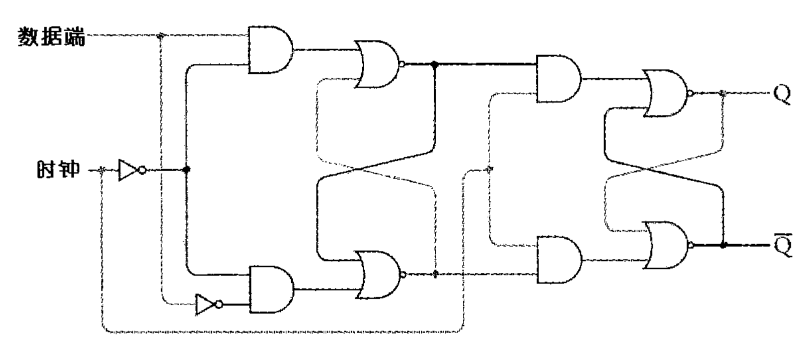
进一步分析，下图为一个处于非工作状态的触发器，其数据输入和时钟输入均为 0，且 Q 输出也为 0。



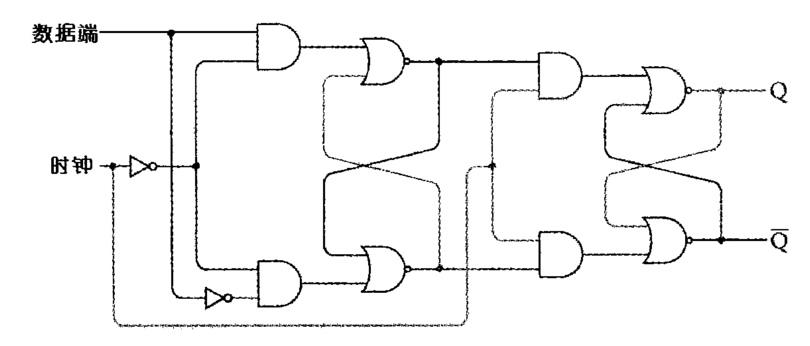
现在使数据端输入为 1，如下图所示。



这改变了第一级触发器的状态，因为时钟输入取反变为 1。但第二级触发器状态保持不变，因为时钟输入仍然为 0。现在把时钟输入变为 1。



这就引起了第二级触发器输出的改变，使 Q 输出变为 1。不同点在于，无论数据端输入发生何种变化（比方说变为 0）都不会影响 Q 的输出。

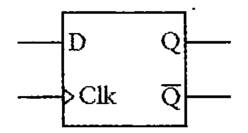


只有在时钟输入从 0 变化到 1 的瞬间 Q 和 $\overline{Q}$ 输出才发生变化。

边沿触发的 D 型触发器的功能表需要一个新的符号来表示从 0 到 1 的瞬时变化，即用一个向上的箭头（↑）表示，如下表所示。

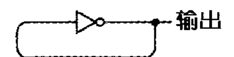
输入		输出	
D	Clk	Q	$\overline{Q}$
0	↑	0	1
1	↑	1	0
X	0	Q	$\overline{Q}$

表中箭头表示当时钟端由 0 变为 1 时（称为时钟信号的“正跳变”，“负跳变”是指从 1 变为 0），Q 端输出与数据端输入是相同的。触发器的符号如下图所示。

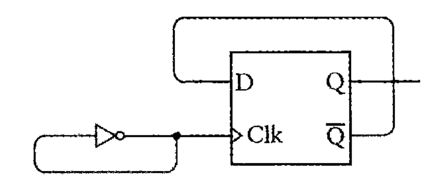


图中的小三角符号表示触发器是边沿触发的。

下面展示的是一个使用边沿 D 型触发器的电路，这个电路是不能用电平触发形式复制出来的。先回忆一下本章开始构造的振荡器，其输出在 0 和 1 之间变化。



把振荡器的输出与边沿触发的 D 型触发器的时钟端输入连接，同时把 $\overline{Q}$ 端输出连接到本身的 D 输入端。



这个触发器的输出同时又是它自己的输入。反馈紧接着反馈！（实际上，这种构造可能是有问题的，振荡器是由状态来回迅速改变的继电器构成的，其输出与构成触发器的继电器相连，而这些其他的继电器不一定能跟得上振荡器的速度。为了避免这些问题，这里假设振荡器中的继电器比电路中其他地方的继电器速度要慢得多）

仔细看一看下面的功能表就可以明白在电路中发生的情况了，电路启动时，假设时钟输入为 0 且 Q 输出也为 0，则 $\overline{Q}$ 端输出为 1，而 $\overline{Q}$ 是和 D 端输入相连的。

输入		输出	
D	Clk	Q	$\overline{Q}$
1	0	0	1

当时钟输入从 0 变为 1 时，Q 输出与 D 输入相同。

输入		输出	
D	Clk	Q	$\overline{Q}$
1	0	0	1
1	↑	1	0

但是由于 $\overline{Q}$ 的输出变为 0，因此 D 输入也变为 0。现在时钟输入为 1，如下表所示。

输入		输出	
D	Clk	Q	$\overline{Q}$
1	0	0	1
1	↑	1	0
0	1	1	0

当时钟输入变回 0 时，不会影响到输出，如下表所示。

输入		输出	
D	Clk	Q	$\overline{Q}$
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0

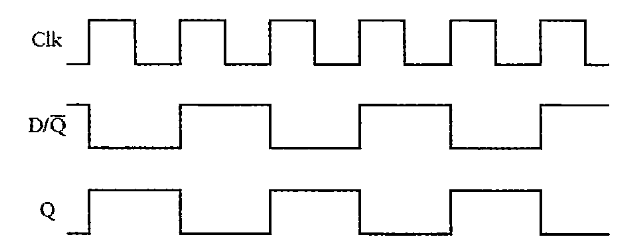
现在时钟端输入又变为 1。由于 D 输入为 0，那么 Q 输出为 0 且 $\overline{Q}$ 输出为 1。

输入		输出	
D	Clk	Q	$\overline{Q}$
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0
0	↑	0	1

所以 D 输入也变为 1，如下表所示。

输入		输出	
D	Clk	Q	$\overline{Q}$
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0
0	↑	0	1
1	1	0	1

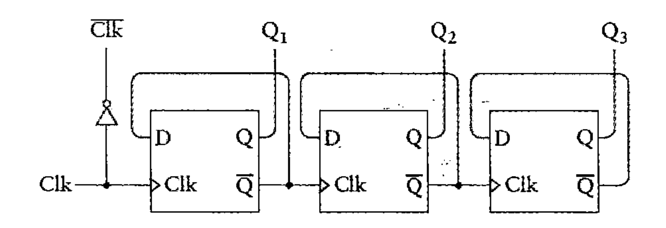
以上发生的现象可以简单总结为：每当时钟输入由 0 变为 1 时，Q 端输出就发生变化，或者从 0 到 1，或者由 1 到 0。下面的时序图可以更加清楚地说明这个问题。



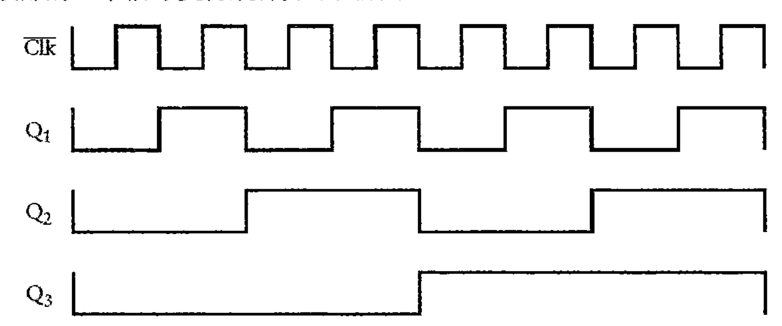
当时钟端 Clk 输入由 0 变为 1 时，D 的值（与 $\overline{Q}$ 的值相同）被输出到 Q 端。当下一次 Clk 信号由 0 变为 1 时，D 和 $\overline{Q}$ 的值同样会改变。

如果这个振荡器的频率是 20Hz（即 20 个周期的时间为 1s），那么 Q 的输出频率是它的一半，即 10Hz，由于这个原因，这种电路称为分频器（frequency divider），它的 $\overline{Q}$ 输出反馈到触发器的数据端输入 D。

当然，分频器的输出可以作为另一个分频器的 Clk 输入，并再一次进行分频。下面是三个分频器连接在一起的示意图。

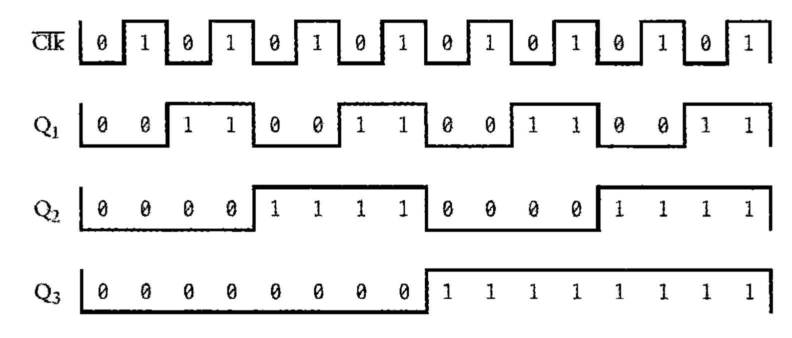


上图顶部的 4 个信号变化规律如下图所示。



这里只给出了这幅图的一部分，因为这个电路会重复上述过程周而复始地变化下去。在这幅图中，你有没有发现眼熟的东西呢？

提示一下，把这些信号标上 0 和 1。



现在看出来了么？试着把这个图顺时针旋转 90° ，然后读一读每一行的 4 位数字，它们分别对应了十进制中的 0～15 中的一个数。

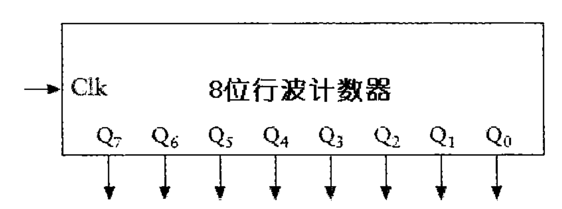
二进制	十进制
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

续表

1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

可以看出，这个电路不仅仅具备了一个计数功能。当然，如果在这个电路中添加更多的触发器，其计数范围就会更大。在第 8 章中提到一个顺序递增的二进制序列，每一列数字在 0 和 1 之间的变化频率是其右边那一列数字变化频率的一半，这个计数器就是模仿了这一点。在每一次时钟信号的正跳变时，计数器的输出是增加的，即递增 1。

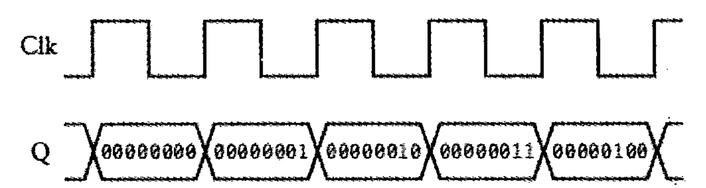
把 8 个触发器连接在一起，然后放入一个盒子中，构成了一个 8 位计数器。



这个计数器称为“8 位行波计数器”，因为每一个触发器的输出都是下一个触发器的时钟输入。变化是在触发器中一级一级地顺序传递的，最后一级触发器的变化必定会有一些延迟，更先进的计数器是“并行（同步）计数器”，这种计数器的所有输出是在同一时刻改变的。

在计数器中输出端用 $Q_0 \sim Q_7$ 标记，在最右边的 Q_0 是第一个触发器的输出。如果将灯泡连到这些输出端上，就可以将 8 位数字读出来。

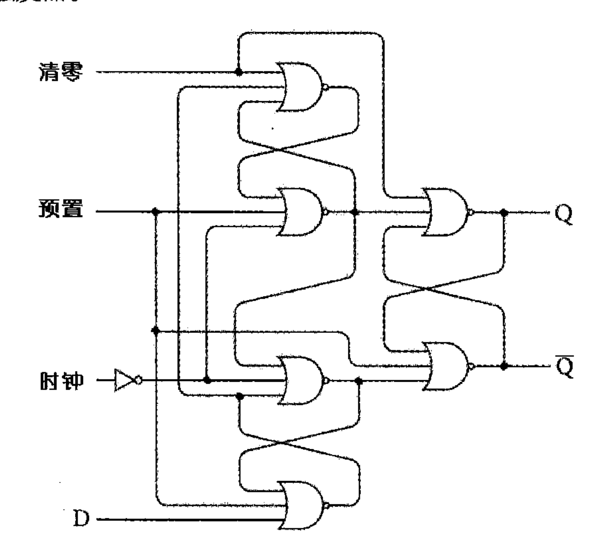
这样一个计数器的时序图可以将 8 个输出分别表示出来，也可以将它们作为整体一起表示出来，如下图所示。



时钟信号的每一个正跳变发生时，一些 Q 输出可能会改变，而另外一些可能不变，但总体上来说它们所表示的二进制编码递增了 1。

本章前面提到过可以找到某种方法来确定振荡器频率，现在已经找到这种方法了。如果把一个振荡器连接到 8 位计数器的时钟输入端上，那么这个计数器会显示出振荡器经过的循环次数。当计数器总数达到 11111111（十进制的 255），它又返回为 00000000。使用计数器确定振荡器频率的最简单的方法就是把计数器的 8 个输出端分别接到 8 只灯泡上。当所有的输出都是 0 时（即所有灯泡都是熄灭的），启动一个秒表计时；当所有灯泡都点亮时，停止秒表计时。这就是振荡器循环 256 次所需要的时间。假设这个时间为 10s，则振荡器的频率是 256 ÷ 10，即 25.6 Hz。

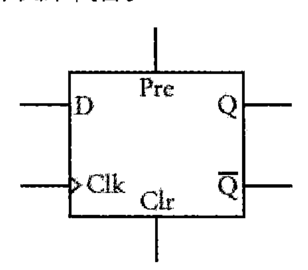
随着触发器功能的增加，它的结构也变得更加复杂，下面给出了一个带预置和清零功能的边沿型 D 触发器。



通常情况下，预置和清零信号输入会覆盖时钟和数据端输入，且两个输入都为 0，当预置信号为 1 时，Q 变为 1，$\overline{Q}$变为 0。当清零信号为 1，Q 变为 0，$\overline{Q}$变为 1（同 R-S 触发器中的 S 和 R 输入端一样，预置和清零信号不能同时为 1）。除此之外，该触发器工作原理是和普通边沿 D 触发器是一样的。

输入				输出	
Pre	Clr	D	Clk	Q	$\overline{Q}$
1	0	X	X	1	0
0	1	X	X	0	1
0	0	0	↑	0	1
0	0	1	↑	1	0
0	0	X	0	Q	$\overline{Q}$

电路图符号可以简单地用下图来代替。



现在，我们已经懂得如何使用继电器来做加法、减法和计数了，这是一件很有成就感的事情，因为我们使用的硬件是 100 多年前就存在的东西。我们还有更多未知领域要去探索，请稍事休息，停止思考那些构造方面的问题，回过头再来看看数字方面的问题吧。

15

字节与十六进制

通过对上一章中两类改进加法机的剖析，我们学习了数据路径（Data Path）这个概念。纵观整个电路的脉络，每 8 个比特流为一组，如潺潺溪流般在器件与器件之间流动。其实，这 8 位比特流就是加法器、锁存器以及数据选择器的输入形式，同时它也是这些器件单元的输出形式。这 8 位的比特流可以用开关的不同状态组合所定义，而且可以用灯泡的亮灭来显示。这样一来，这些电路中数据路径的位宽（bits wide）就是 8。为什么我们要把它定义为"8"位呢？为什么没有定义为 6 位、7 位、9 位或 10 位呢？

要想用偷懒的方式回答上面这个问题也很简单，答案就是这些加法机都是对第 12 章中原始加法机的改进，而它所"流传"下来的位宽恰好就是 8 位。但是追根究底，原始加法机偏偏是 8 位其实没有什么特别的原因。只不过在每次使用 8 位位宽时，一切工作都显得非常方便——一种优雅的比特化（biteful）的比特流。或许大家已经感觉到我在极力隐瞒一些东西，无法否认，其实我心中一直都很清楚（或许你也知道）：8 比特代表一个字节（byte）。

字节这个词最早起源于 1956 年前后，由 IBM 公司提出。最早的拼写方式是 bite，但为了避免与 bit 混淆用 y 代替了 i。曾几何时，字节仅表示某一数据路径上的位数，直到 20 世纪 60 年代中叶，在 IBM 的 360 系统的发展下（一种大规模复杂的商用计算机），字节这个词逐渐开始用来表示一组 8 比特数据。

由于有 8 位，一个字节的取值范围为 00000000 到 11111111。相应地，它还可以表示成为 0～255 之间的正整数，如果将一个数的补码作为其相对应的负数，那么一个字节可以表示在–128～127 范围内的正、负整数。一个给定的字节可以代表 2^8，即 256 种不同事物中的一个。

让我们再来看看“8”这个数字，当 8 作为比特流的一种尺度，它的确表现出了非常完美的特质。字节在很多方面都比单独的比特更胜一筹。IBM 采用字节有一个很重要的原因，就是这样一来数字就可以按照 BCD 形式（第 23 章中将会讲述）方便地保存。在本章随后的讲述中，我们会发现凑巧的是：全世界大部分书面语言（除了中文、日文以及韩文中使用的象形文字体系）的基本字符数都少于 256，所以字节是一种理想的保存文本的手段。字节同样适合表示黑白图像中的灰度值，这是由于肉眼能区分的灰度约为 256 种。当一个字节无法表示所有信息（如刚提到的中文、日文以及韩文中使用的象形文字体系等），我们只需采用两个字节——就可以表示 2^{16} 也就是 65536 个不同的物体——这也是一种很好的解决方案。

字节的一半——即 4 比特——我们称之为半字节（nibble，也可拼写成 nybble），在计算机这个领域，它并不像字节那样经常使用。

由于字节在计算机内部出现的频率较高，如果可以使用一种简洁的方式将它的内在含义准确表达出来，将会为我们带来很多方便。假如一个 8 位二进制数表示为 10110110，这种表达方式自然而又直观，但它还不够简洁。

我们完全可以采用十进制表示法来表示字节，但从二进制转换到十进制需要进行一系列计算——计算方法并不复杂，但是比较麻烦。我曾在第 8 章曾介绍过一种直观的计算方法。因为每一位二进制数对应着 2 的不同幂，因此我们可以把二进制数写到第一行，在底下写出每一位二进制数对应的 2 的乘方数，然后采用“列相乘、行相加”就可以得到对应的十进制数。下图表示了 10110110 的转换过程。

1	0	1	1	0	1	1	0
×128	×64	×32	×16	×8	×4	×2	×1

128 + 0 + 32 + 16 + 0 + 4 + 2 + 0 = 182

把十进制数转换为二进制数就需要一点点技巧了。我们可以用这个十进制数不断除以递减排列的 2 的幂，每除一次得到的商就是所要求的二进制数中的一位，而余数成为

下一次运算的除数，它的作用是除以下一个较小一点的 2 的幂。下图表示了十进制数 182 转换成二进制形式的过程。

182	54	54	22	6	6	2	0
÷128	÷64	÷32	÷16	÷8	÷4	÷2	÷1
1	0	1	1	0	1	1	0

第 8 章对这种计算方法有更详细的描述。不过，在十进制数和二进制数之间进行转换还需用到一些工具，例如要借助笔和纸进行一系列演算。

第 8 章中我们还学到了八进制数，也称为八进制数字系统。这种系统仅使用到了数字 0、1、2、3、4、5、6 还有 7。八进制数和二进制数之间的转换简洁方便，只要记住 0～7 这 8 个数字所对应的 3 位二进制数即可。下面这张表就表示了这种对应关系。

二进制数	八进制数
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

假如要把一个二进制数（如 10110110）转化为 8 进制，可以从最右端的数字开始。每 3 比特看做一组，这样每组便对应着一个八进制数：

$$\underbrace{10}_{2}\underbrace{110}_{6}\underbrace{110}_{6}$$

10110110 这个字节很容易就表示为八进制数 266。这种方法简洁明了，八进制用来表示字节不失为一个好方法。但还是有那么一点美中不足。

字节可以表示的二进制数的范围为 00000000～11111111，如果采用八进制表示，那么相对应的范围也随之变成了 000～377。仔细分析下先前的例子，我们从右到左把 3 位二进制数对应于中间以及最靠右的八进制数，而最靠左的八进制数却是由 2 位二进制数对

应的。如果我们将 16 位二进制数直接表示为八进制会得到如下结果：

```
1011001111000101
1 3  1  7  0  5
```

如果我们把这个 16 位二进制数平分为两个字节并将其分别表示为八进制数会得到如下所示的不同结果：

```
10110011    11000101
2 6  3      3 0  5
```

为了使多字节值能和分开表示的单字节值取得一致，我们需要一种可以等分单个字节的系统，按照这种思想，我们可以把每个字节等分成 4 组，每组 2 比特（基于 4 的计数系统）；还可以等分为 2 组，每组 4 比特（基于 16 的计数系统）。

基于 16 的计数系统（Base 16），对于我们而言是一种全新的系统。基于 16 的计数系统也被称为十六进制（hexadecimal），这个单词很容易让我们产生混淆。这是因为很多以 hexa-为前缀的单词（比如 Hexagon， Hexapod 以及 Hexameter）都会与 6 或多或少有些联系。而 hexadecimal 这个词却偏偏代表了 16（Sixteen）这个含义。虽然在《微软出版物风格与技术手册》(*The Microsoft Manual of Style for Technical Publications*)中明确说明“请勿将十六进制缩写为 hex”，但包括我在内的绝大多数人还总是在不经意间使用这种缩写。

十六进制还有其他一些特别之处。比如在十进制中，我们通常可以用下面这种方式计数：

0 1 2 3 4 5 6 7 8 9 10 11 12...

我们仔细回忆一下在八进制中，8 和 9 这两个数字是不需要的，就像下面这样：

0 1 2 3 4 5 6 7 10 11 12...

同理，四进制计数中我们也不需要 4、5、6、7 这些数字，就像下面这样：

0 1 2 3 10 11 12...

在二进制中，这一切变得更加简单，我们所需要的只有 0 和 1：

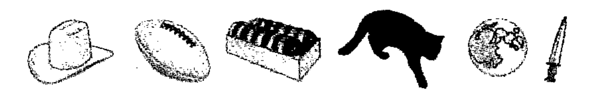
0 1 10 11 100...

十六进制计数和上面所讲的这一系列情况是完全不同的，它需要比十进制更多的数

字来计数。十六进制的计数过程将会是下面这种形式：

0 1 2 3 4 5 6 7 8 9 ? ? ? ? ? ? 10 11 12...

上图中 10（准确地来说是一个 1 紧挨着一个 0）这个数字代表的准确含义实际上应该是十进制中的 16。图中的问号表明我们还需要 6 个符号来完整表示十六进制系统。这些符号究竟是什么？它们出于何处？仔细想想，原有的计数系统中的符号都有自己的唯一身份，所以应该而且必须引入新的符号。寻找 6 个新符号对我们来说是小菜一碟，例如下面这几个符号：



不像我们所使用的大多数数字符号，上面列出的每个符号都很容易识记，而且其背后都隐含着实际数字意义，这种形象的符号使得它们便于记忆。比如这个 10 加仑的牛仔帽、一个橄榄球（11 个人组成一支橄榄球队）、一打（12 个）面包圈、一只黑猫（使人们想起不吉利的 13）、一轮满月（一般出现在后弦月 14 天之后的夜晚），一把匕首让人们联想到凯撒大帝（Julius Caesar）在三月的月中（第 15 日）被刺杀。

两个十六进制数可以完整地代表一个字节。这也意味着一个十六进制数恰好由 4 位二进制数组成，即半字节。下面这张表描述了如何在二进制、十六进制、十进制数之间进行转换。

二进制	十六进制	十进制	二进制	十六进制	十进制
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010		10
0011	3	3	1011		11
0100	4	4	1100		12
0101	5	5	1101		13
0110	6	6	1110		14
0111	7	7	1111		15

让我们再来看看如何把字节 10110110 转换成十六进制。

10110110

6

无论是多字节还是单字节，我们都可以进行转换。

10110110　　11000101

6　　5

一个字节能且只能由一对十六进制数来表示。

只可惜（或许你也松了口气）我们不打算用橄榄球或一打面包圈来表示十六进制数。虽然这种方案完全可行，但它不是很方便也不够正规，有时还会让人感到迷惑。事实上十六进制中缺少的 6 个符号由 6 个拉丁字母来表示，就像下面这样：

0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12...

下面这张表描述了正规的二进制、十六进制、十进制转换的过程。

二进制	十六进制	十进制
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

这样一来字节10110110就可以表示为十六进制的B6,而不用画一个橄榄球那么麻烦。我们再回忆回忆前面的章节，下面举一个完整的进制转换的例子（用下标代表进制），比如：

10110110_{TWO}（10110110_2）

它的四进制形式表示为：

2312_{FOUR}（2312_4）

它的八进制形式表示为：

266_{EITHT}（266_8）

它的十进制形式表示为：

182_{TEN}（182_{10}）

与此类似，它的十六进制形式可以表示为：

$B6_{SIXTEEN}$（$B6_{16}$）

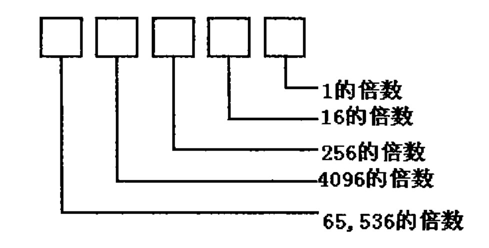
但是这种写法有点累赘，幸好还有几种表达十六进制数通用方法。你还可以用如下方法表示：

$B6_{HEX}$

在本书中将使用一种更加简洁实用的方法，那就是用一个小写的 h 紧跟在数字后边表示这个数是以十六进制表示的，就像这样：

B6h

通过分析可以得到十六进制数的每一位代表 16 的不同整数幂的倍数，如下图所示。



十六进制数 9A48Ch 可以表示为如下形式：

$$\begin{aligned} 9A48Ch = 9 \times 10000h &+ \\ A \times 1000h &+ \\ 4 \times 100h &+ \\ 8 \times 10h &+ \\ C \times 1h & \end{aligned}$$

这个数用 16 的乘方表示可以写为：

$$\begin{aligned} 9A48Ch = 9 \times 16^4 &+ \\ A \times 16^3 &+ \\ 4 \times 16^2 &+ \\ 8 \times 16^1 &+ \\ C \times 16^0 & \end{aligned}$$

也可以将 16 的幂进一步展开，写为如下形式：

$$\begin{aligned} 9A48Ch = 9 \times 65{,}536 &+ \\ A \times 4096 &+ \\ 4 \times 256 &+ \\ 8 \times 16 &+ \\ C \times 1 & \end{aligned}$$

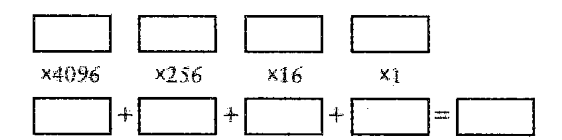
我们可以仔细想想，如果只把一个数字（比如 9，A，4，8 和 C 中的任何一个）单独列举出来，而且不使用任何下标来指明其进制数，这种做法并不会产生二义性。无论在十进制还是十六进制下，一个单独的 9 仅代表 9；而 A 的出现也说明了它本身是一个十六进制数——等价于十进制中的 10。

我们可以拿起笔和纸进行演算了，把十六进制数转换成十进制数：

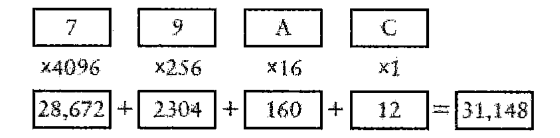
$$\begin{aligned} 9A48Ch = 9 \times 65{,}536 &+ \\ 10 \times 4096 &+ \\ 4 \times 256 &+ \\ 8 \times 16 &+ \\ 12 \times 1 & \end{aligned}$$

运算的最后结果是 631,948。这就是一个十六进制数转换成为十进制数的完整过程。

下面给出了一种模板，它可以帮助我们把 4 位十六进制转换成为十进制。

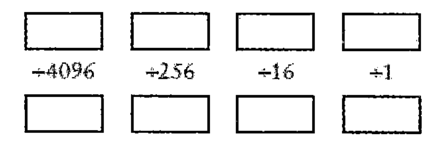


下面我们来看一个例子，把 79ACh 转化成为十进制数。需要牢记在心的就是 A 和 C 分别代表 10 和 12。

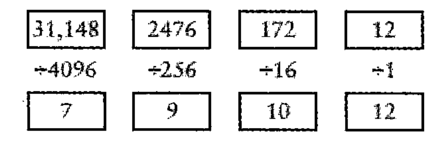


十进制数转换为十六进制数通常涉及除法运算。我们知道，小于或等于 255 的数用 1 个字节就足以表示，也就是两个十六进制数。如何来求出这两个数呢？通常可以用这个数除以 16，分别得到商和余数，商作为结果保留，而余数则作为下次运算的除数。我们举一个先前讲过的例子来进一步阐述运算法则：十进制数 182，除以 16，商为 11（在十六进制中表示为 B），余数为 6，所以它的十六进制为 B6h。

如果被转化的十进制数小于 65,536，那么就可以使用少于 4 位的十六进制数来表示。下面给出一个十进制向十六进制数转化的一个模板。



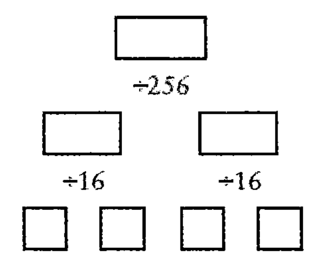
使用时首先把十进制写到左上角的方框里。这个方框代表着第一个被除数，然后除以第一个除数 4096，得到的商放到被除数所对应的下面的方框里，而将余数放到被除数右边的方框里。这时将余数作为新的被除数去除以 256。利用这个规则，通过反复迭代就可以得到最终结果。下面这幅图向我们展示了十进制数 31,148 转换成十六进制数的过程。



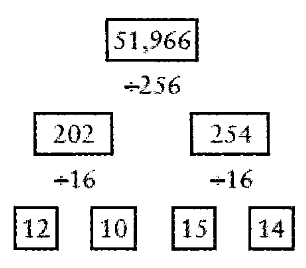
显而易见，十进制数的 10 和 12 代表着十六进制中的 A 和 C，计算得到的最后结果就是 79ACh。

这个方法存在一个小小的问题，那就是如果使用计算器帮助你进行除法运算，它不会显示每步运算得到的余数是多少。如果让计算器去运算 31,148 除以 4096，它的屏幕上只会出现 7.6044921875。我们可以用倒推的方法计算余数，用 4096 乘以 7（得到 28,672），从 31,148 中减去它，这是一种办法。也可以用 4096 乘以 0.6044921875，也就是计算器运算结果的小数部分（现在市面上很多计算器已经具备了十进制和十六进制数之间的转换功能）。

还有一种转换小于 65,535 的十六进制数的方法，首先我们把原数通过除以 256 的方式将其分为两个字节。接下来对于每个字节，再分别除以 16。下图是运算过程使用到的模板。



我们采用自顶向下的方式。每一次除法完成后，就将其得到的商放入除数左下方的方框里，而余数进入右边的方框里。下图举例说明了十进制数 51,966 的转换过程。



最后我们得到了四个 1 位的十六进制数字，其十进制值分别是 12、10、15 和 14，转换过来就是 CAFE，无论怎么看它都更像一个单词，很难想象它其实是一个数字（我想应该没人愿意点一杯叫做 56,495 的东西，然后把它当作咖啡喝下去吧）!

对于每种计数系统，我们都可以描绘出相应的操作运算表，下面是十六进制的加法运算表。

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

使用这张表可以方便地仿照一般的加法运算来对十六进制数进行加运算，如下例所示：

$$\begin{array}{r} \text{4A3378E2} \\ +\ \text{877AB982} \\ \hline \text{D1AE3264} \end{array}$$

回忆一下第 13 章的内容，我们讨论过可以用一个 2 的补数来表示与其相对应的负数。如果我们处理的是带符号的 8 位二进制数，那么所有负数的最高位都为 1。在十六进制系统中，最高位为 8、9、A、B、C、D、E 或 F 的两位带符号数都是负数，因为这些十六进制数对应的二进制数的最高位为 1。例如 99h 可以表示无符号的十进制数 153（你必须清楚它是单字节的无符号数），也可以表示十进制的–103（这时它被看做有符号数）。

奇妙的是，99h 这个十六进制字节从某种意义上来讲，也代表着十进制的 99！我们都非常想知道原因，不过这种说法似乎和先前所学到的所有东西相抵触。我会在第 23 章中进行详细解释。但在此之前，我们必须先讨论一下存储器方面的知识。

16 存储器组织

每天清晨，我们将自己从沉睡中唤醒，这时大脑的空白会很快被记忆填充。我们立刻会意识到自己身在何方，最近做了些什么事情，有什么计划和打算。有的事情我们很快就能想起来，但有时，我们大脑处于失忆状态，有那么几分钟发现自己什么都想不起来（就拿我来说，有时我就是想不起来怎么我上床时还穿着袜子），但总的来说，我们总是能够与自己的过去保持足够的连续性，继续新的生活，展开人生新的一页。

显然，人类的记忆似乎没有什么规律。仔细回想高中的几何课，或许你一下子就能想到是谁坐在你前面，或许你清晰地记得当老师讲到 QED（quod erat demonstrandum，证明与推论）这个概念的时候消防演习开始了。

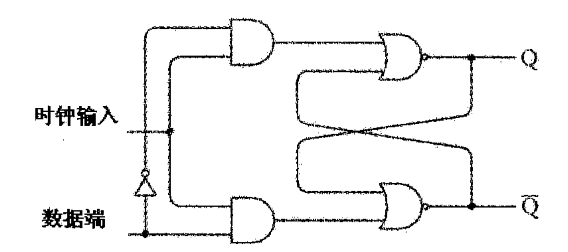
人类的记忆也并非能面面俱到。书面记录这种技术的引入，从某种层面来讲，就是为了弥补人类记忆容易遗漏这一缺陷。或许在某天夜里凌晨三点，你从床上一跃而起，脑海中突然出现某个剧本的绝妙灵感。你立刻抓起床边提前预备的笔和纸，将它们全部记下防止遗忘，然后才安然入睡。一觉醒来的清晨，再次浏览这个绝妙的灵感，一个新的剧本构想跃然纸上（剧本的内容就是“一次邂逅，汽车追尾与爆炸”?仅此而已吗？）或许远不如此。

我们总是将需要的记住的内容事先记下来，在需要时拿出来阅读；习惯于将可能用

到的事物先存起来，在需要时将它们取出。从技术角度来讲，这个过程称为先存储后访问。存储器的职责和作用就在于此，它负责保障这两个过程之间信息完好无损。我们每次存储信息都要利用不同种类的存储器。比如，保存文本信息的不二之选就是纸张，而磁带则更适于存储音乐和电影。

电报继电器（Telegraph Relays）——以一定形式组织起来构成逻辑门，然后再形成触发器——同样具备保存信息的能力。在前面章节中我们讨论过，一个触发器可以对 1 位信息进行存储。这样的存储能力要存储一大堆的信息还远远不够，但它却为我们达到目标迈出了坚实的一步。其实知道了如何存储 1 位信息，很容易就可以想象出如何存储 2 位、3 位或更多位信息。

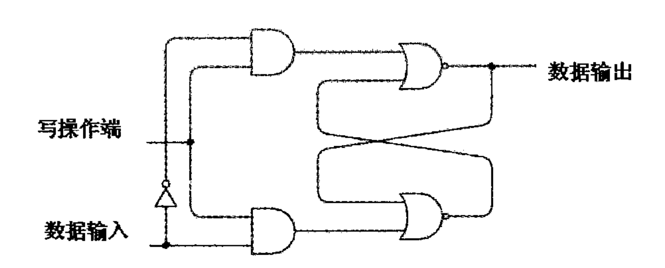
在第 14 章的学习过程中，我们一起讨论过由一个反向器、两个与门和两个或非门构成的 D 型电平触发器，如下图所示。



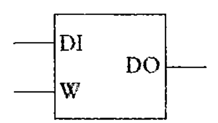
当时钟输入为 1，Q 端输出与数据端输入保持一致。但当时钟输入跳变为 0 时，Q 端输出将保持数据端最后一次的输入。除非时钟输入再次还原为 1，之后的数据端输入不会影响输出。此触发器的真值表如下。

输入		输出	
D	Clk	Q	$\overline{Q}$
0	1	0	1
1	1	1	0
X	0	Q	$\overline{Q}$

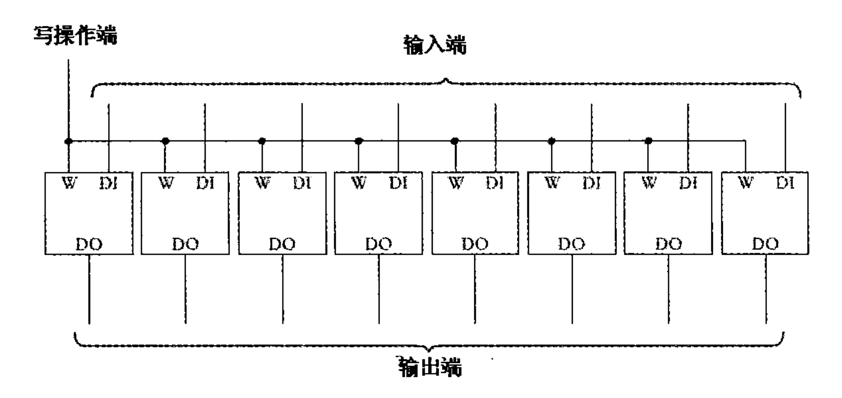
在第 14 章的讨论中，这种触发器可以由两种不同特性的电路来实现，而在本章我们仅选择其中一种——目的就只是为了保存 1 位信息。为了更加清楚地表述，我们给输入和输出端重新命名，使其名称与功能相符，如下图所示。



从结构上来讲，这套电路与先前所学到的是同一种触发器，只是命名的方式不尽相同，现在 Q 输出端被称为数据输出端（Data Out），时钟输入端（在第 14 章叫做保持位）命名为写操作端（Write）。就像信息可以被记录在纸上一样，写操作端的信号同样使得数据输入（Data In）信号被写入（Written Into），也可以称之为被存储（stored）到电路中。一般情况下，如果写操作端为 0，则数据输入信号的状态对输出无影响。而当我们想把数据输入信号存储在触发器中时，可以把写入信号应先置 1 后置为 0。在第 14 章讲到过，这种类型的电路也被称为锁存器，因为存储进去的数据就好像被锁住了一样。下面给出了 1 位锁存器简化框图，框图未画出其内部结构中的部件。

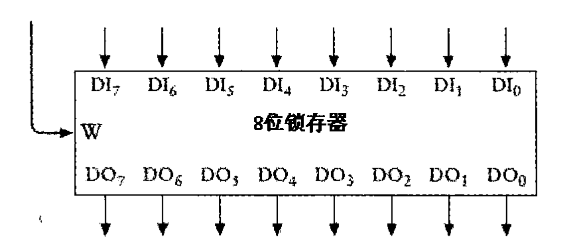


我们很容易想到如何把多个 1 位锁存器组织成为多位锁存器，所要做的就是把写操作端的信号连接到系统中，就像下面这样。

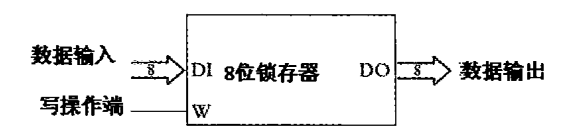


图中显示的 8 位锁存器其输入和输出端各有 8 个。另外还包括一个写操作端，在非工作状态下一般为 0。如果要把一个 8 位二进制数存储在锁存器中，首先要把写操作端置 1，

然后置 0。我们同样可以把这个锁存器以框图的形式表现出来，就像下面这样。



为了和先前提到的 1 位锁存器保持一致，我们将它可以画成下面这种形式。



还有另一种方法集成 8 个 1 位锁存器，但其结构并不像上面的这样直观。假设我们只想用一个数据输入和输出信号端，而且希望锁存器能将输入信号数据分 8 次独立存储，这个任务可以在长达一天内完成，或者可能迅速在下一分钟内搞定。最后一项要求就是我们还希望能够通过观察数据输出信号端确定实际的 8 位输出。

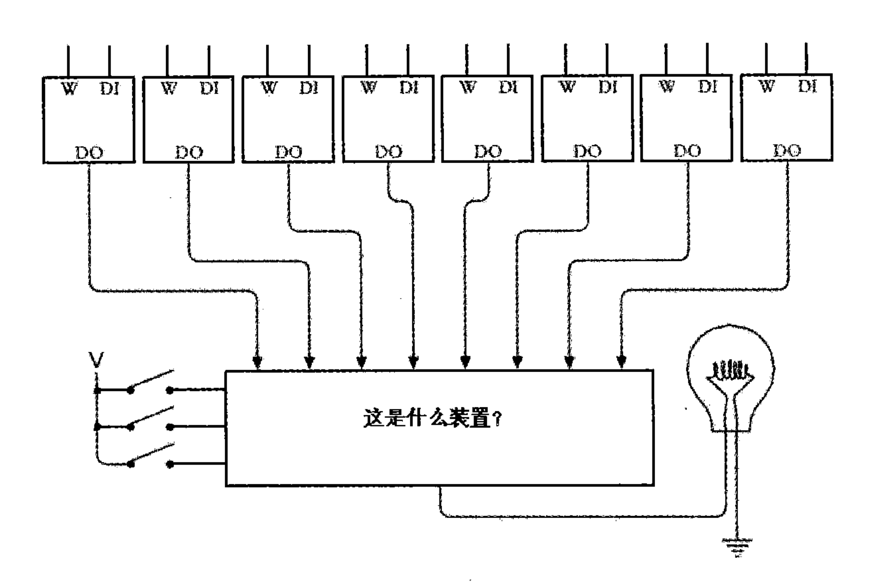
换句话说，在这种锁存器中我们只想存储 8 个单独的比特，而不是存储 1 个 8 位二进制数。

为什么会有这种需求呢？原因可能在于仅有一个灯泡！

我们知道现在所需要的是 8 个 1 位锁存器。先不去考虑数据如何存储在这些锁存器中，把重点放在如何用一个灯泡来确定锁存器的数据输出信号。最简单的方法就是把这个灯泡依次连接到每个锁存器上，分若干次来测试各个锁存器的输出，但是我们要追求更加自动化的方法。用开关来选择想要检查的锁存器是一个好的办法。

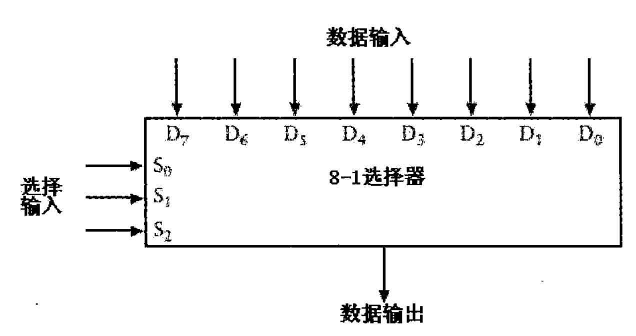
究竟需要多少个开关才能解决问题呢？我们可以把这个过程进一步抽象，问题变成了怎么样从 8 个物体中选出一个我们想要的，我们需要 3 个开关。这是因为通过 3 个开关连通与闭合的排列组合，总共可表示出 8 个不同的值：000、001、010、011、100、101、110 和 111。

现在我们手头上已有 8 个 1 位锁存器、3 个开关、1 个灯泡，此外在开关和灯泡之间还有另外一种装置，如下图所示。



这个“额外装置”就是图中的神秘盒子，顶部带有 8 个输入端，左侧也带有 3 个输入端。通过三个开关的闭合和断开，对顶部的输入进行 8 选 1 操作，输出结果被传递到其底部连接的灯泡，使其发光。

上图标注的“这是什么装置？”到底是什么呢？我们先前曾碰见过类似的东西，当时讨论的装置没有这么多的输入端。这种装置曾出现在第 14 章中第一个改进的加法机的电路中。在该电路中要在一行开关和一个锁存器的输出之间选择一个，作为加法器的输入，当时称其为 2-1 选择器。而我们现在所需要的正是 8-1 数据选择器（8-Line-to-1-Line Data Selector）。

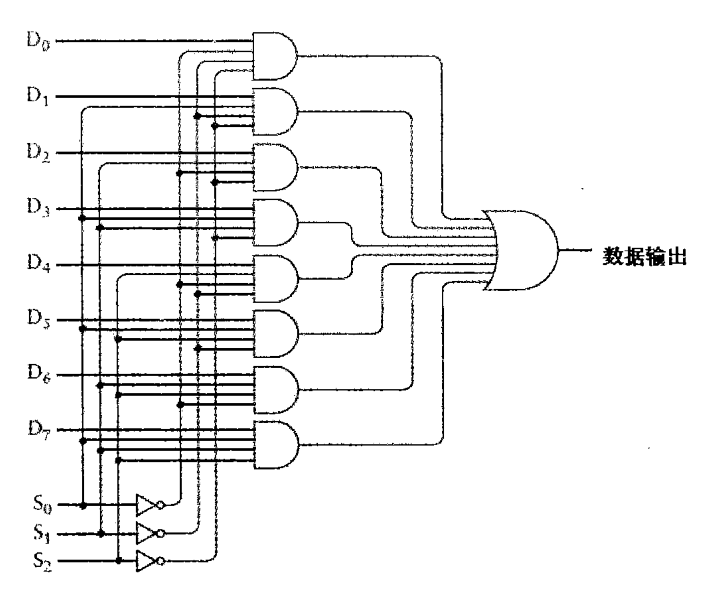


8-1 选择器有 8 个数据输入端（在其顶部），以及 3 个选择输入端（在其左侧）。选择输入端的功能就是选择一个输入端数据，然后使其在输出端输出。如果选择输入端为 000，

则将 D_0 锁存器的值输出；若选择端为 111，则 D_7 锁存器的值将被输出；若选择端为 101，则相应地输出 D_5 的值。系统的真值表如下所示。

输入			输出
S_2	S_1	S_0	Q
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

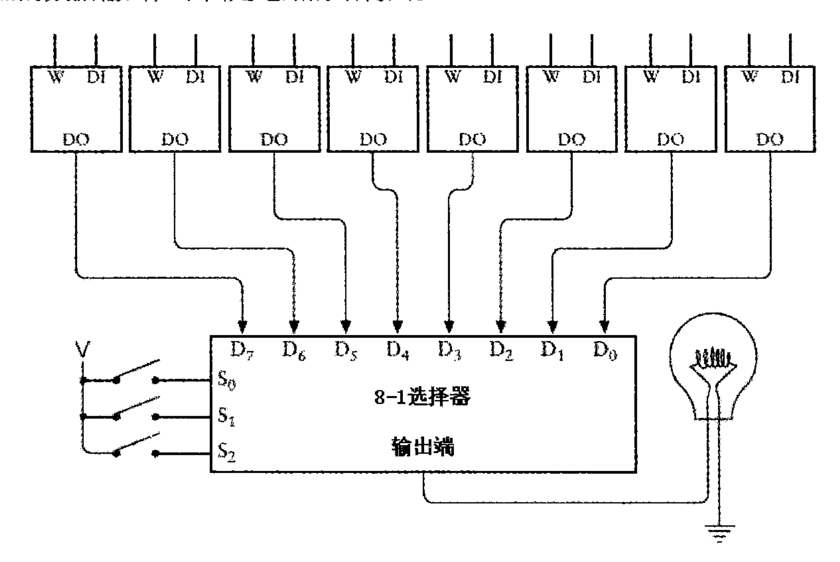
8-1 选择器主要组成部件为：三个反向器、八个 4 端口输入与门、一个 8 端口输入或门，系统的组织结构如下图所示。



这个电路看上去线路密布，要理解它是如何工作的，最好方式就是一起来看一个例子。假设 S_2 初始化为 1，S_1 初始化为 0，S_0 初始化为 1。从顶部开始的第 6 个与门的输入由 S_0、$\overline{S_1}$、S_2 组成，初始状态下它们全为 1。其余与门的这三项输入数据都与第 6 个与

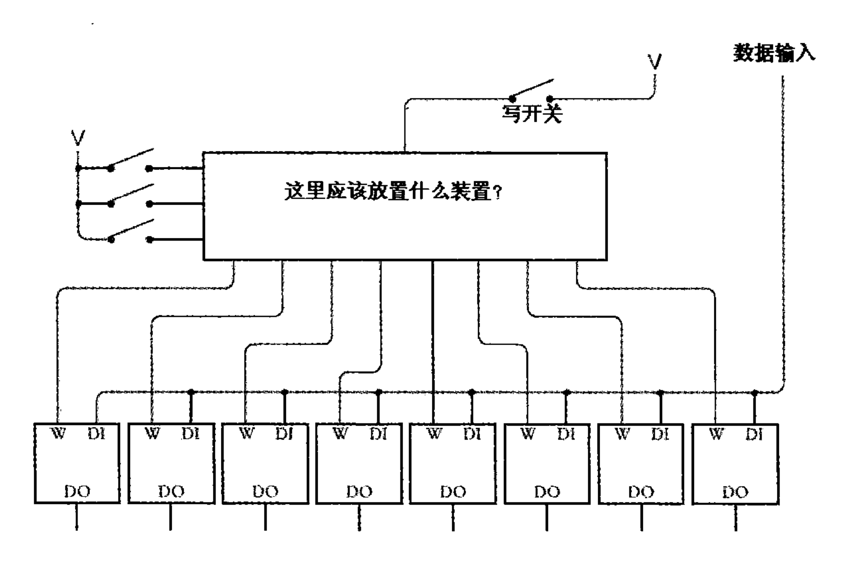
门不尽相同，这使得其余与门输出全部为 0。若 D_5 变为 0 意味着第 6 个与门输出为 0；反之第 6 个与门输出则为 1。对最右边的或门也可以按照同样的方式理解。我们可以总结出下面这个结论：若选择端为 101，则数据输出端与 D_5 的输出保持一致。

让我们重新理一下思路，想想自己究竟要干什么。我们的目的是通过某种方式连接 8 个 1 位锁存器，使自己能够从一个输入信号端写入数据，还能从一个输出信号端鉴别出数据。现在我们已经成功地使用了一个 8-1 选择器对 8 个锁存器进行了选择操作，并将相应锁存器的数据输出，下面是电路的结构图。



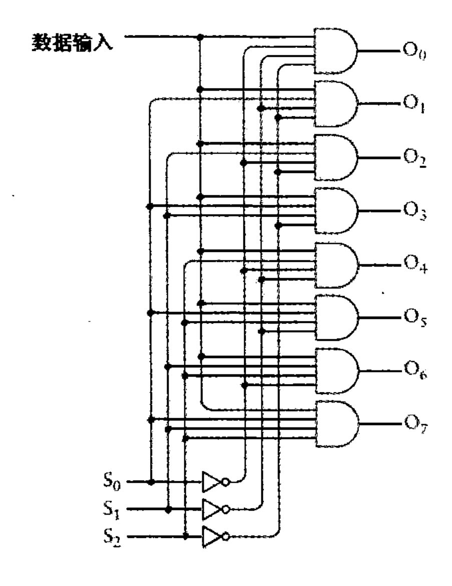
到这里我们只走完了长征的一半。既然输出端已经满足了要求，现在把注意力集中到输入端。

输入端包括了数据输入信号及写操作信号。可以把所有数据输入信号在锁存器的输入端连接在一起。但 8 个写入信号是不可以连在一起的，因为我们很可能要向每个锁存器依次写入数据。除此之外还需要一个独立的写入信号，它能被路由到任意（且唯一）的锁存器上，系统的结构可用下图表示。



为了能圆满完成任务，我们需要另外一款电路元件，而且这款元件与 8-1 选择器功能类似，但它的作用正好相反。我们所说的正是 3-8 译码器（3-to-8 Decoder）。前面的章节中我们曾学习过一个简易的数据译码器（Data Decoder）——在第 11 章中为了选择喜欢的猫咪的毛色，我们把开关以一定方式进行连接使其具有选择功能。

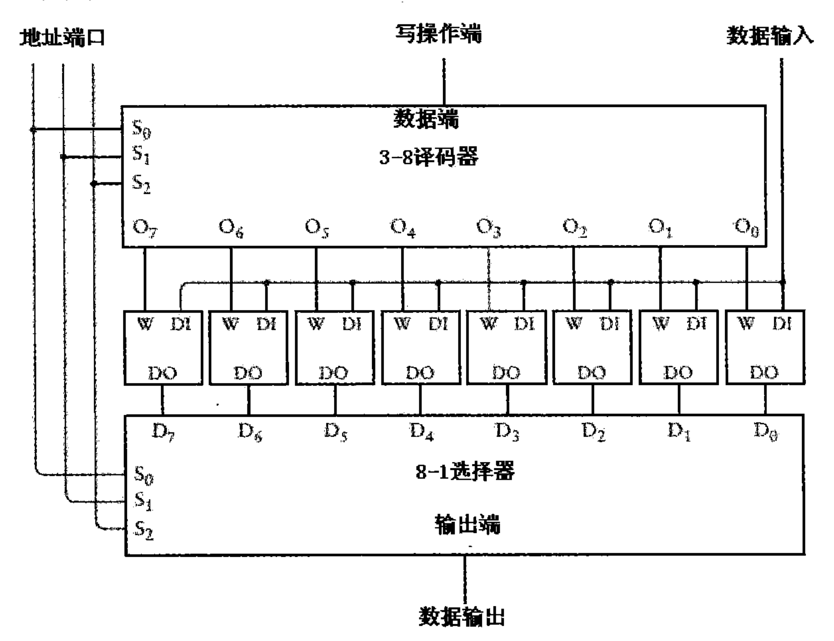
3-8 译码器的输出端口共有 8 个。在任何时刻，译码器只会有一个锁存器的输出为 1，其余均为 0。每一个输出端的结果都是由 S_0、S_1、S_2 这三个信号的排列组合决定的。而数据的输出和输入一致，如下图所示。



我想再次强调一遍：注意从上往下数的第 6 个与门，它的输入包括 S_0、$\overline{S_1}$、S_2。没有任何一个与门具有和它相同的三个输入。在这种情况下，如果选择输入端为 101，则除了 O_5 要根据情况进行判定外，其余与门输出都为 0。这个时候，若数据端输入为 0，则 O_5 随之输出为 0；相应的，若数据端输入为 1，则 O_5 输出为 1。译码器的逻辑表可以如下表所示。

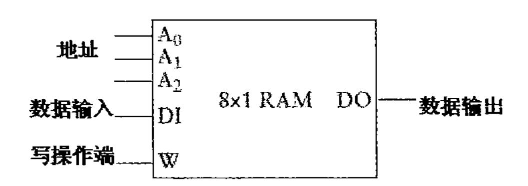
输入			输出							
S_2	S_1	S_0	O_7	O_6	O_5	O_4	O_4	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	0	Data
0	0	1	0	0	0	0	0	0	Data	0
0	1	0	0	0	0	0	0	Data	0	0
0	1	1	0	0	0	0	Data	0	0	0
1	0	0	0	0	0	Data	0	0	0	0
1	0	1	0	0	Data	0	0	0	0	0
1	1	0	0	Data	0	0	0	0	0	0
1	1	1	Data	0	0	0	0	0	0	0

将 8 个锁存器加入到电路就形成了完整的系统。



值得注意的是，译码器和选择器具有相同的选择信号，在上图中这三个信号一起被称为地址端口（Address）。地址的作用就像我们平时使用的邮箱号，长度为三位的地址决定了 8 个锁存器中的哪一个将被引用。在 3-8 译码器的输入端，地址起到了决定哪些锁存器可以被写操作端的信号触发来保存数据的作用。在输出端（图的下半部分），8-1 选择器通过地址来选择 8 个锁存器中的一个，最后将其输出。

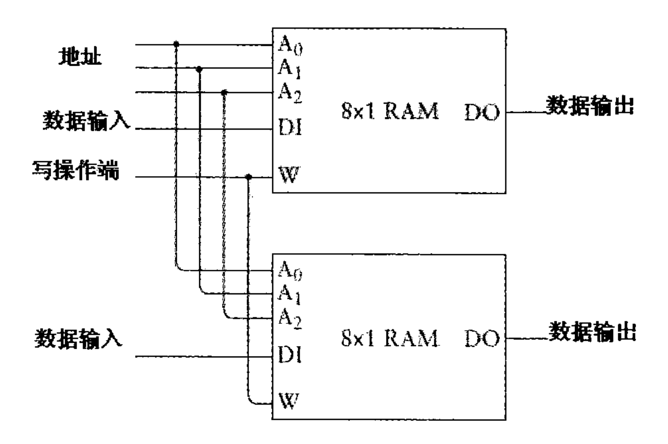
这种配置下的锁存器在有的资料中也被称为读/写存储器（read/write memory），但更普遍的叫法是随机访问存储器（Random Access Memory），或 RAM（和单词 animal 发音类似）。可以认为我们讨论的这种存储器是可存储 8 个独立比特的 RAM，它的简化结构图如下所示。



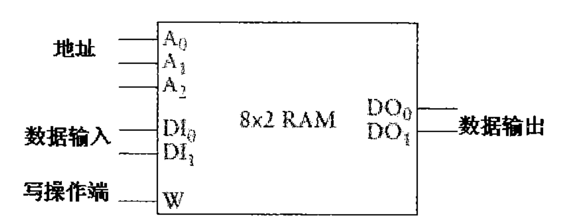
上图所示的电路之所以能够被称为存储器是因为它可以保存信息。而能够被称为读/写存储器是因为它不仅可以在每个锁存器中存储新的数据（可以把这种功能称为写数据），而且我们还可以检查每个锁存器都保存了什么数据（可以把这种功能称为读数据）。之所以可以被称为随机访问存储器，是因为读写操作很自由，我们只需要改变地址及相关的输入，就可以从 8 个锁存器中读出或写入需要的数据。相比于其他的顺序型的存储器——这种存储器在使用时有一定的限制，如果想要读取地址为 101 的数据，必须先把地址为 100 的数据读取出来。

将 RAM 进行特殊的配置可形成 RAM 阵列（Array），我们所讨论的这种 RAM 阵列以 8×1（读做 8 乘 1）的方式组织起来。阵列以 1 比特作为存储单位，共存储 8 个单位的数据。所以这个 RAM 阵列中能存储的位数等于 8 与 1 的乘积。

RAM 阵列的组合形式多种多样。比如我们可以通过共享地址的方式可以把两个 8×1 的 RAM 阵列连接起来，如下图所示。

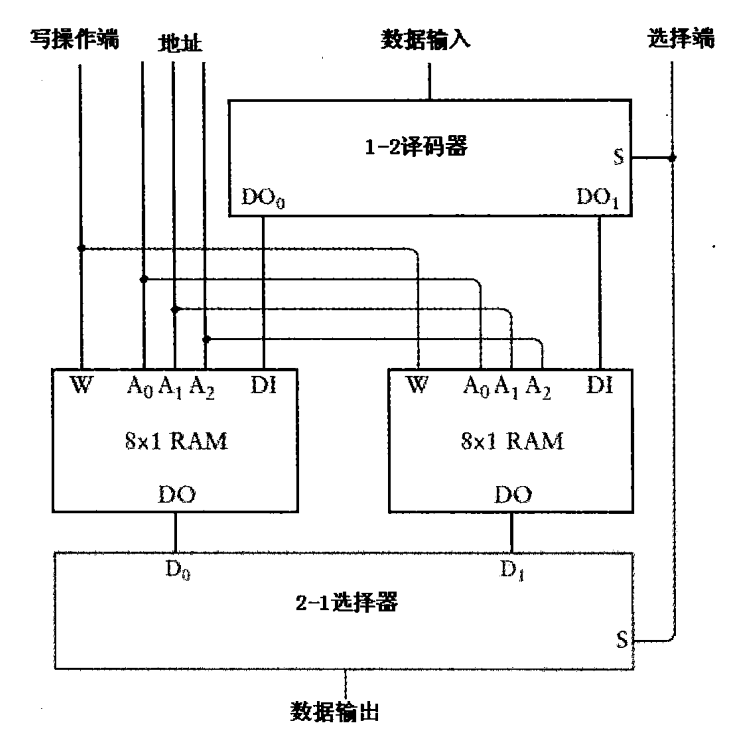


我们把这两个 8×1 的 RAM 阵列的地址和输出都分别看成一个整体，这样就得到了一个 8×2 的 RAM 阵列，如下图所示。

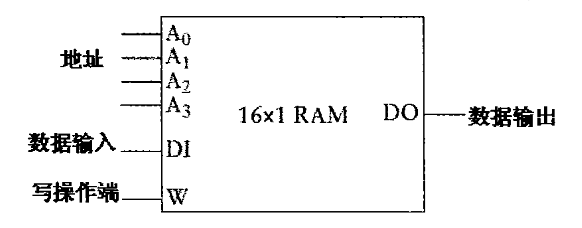


这个 RAM 阵列可存储的二进制数依然是 8 个，但每个数的位宽为 2 位。

我们还可以把两个 8×1 的 RAM 阵列看做是两个锁存器，使用一个 2-1 选择器和一个 1-2 译码器就可以把它们按照单个锁存器连接方式进行集成，下面给出了这种方案的电路图。



“选择”端之所以连接到译码器和选择器，主要作用是在两个 8×1 RAM 阵列中选择一个，本质上它扮演了第 4 根地址线的角色。因此这种结构实质上是一种 16×1 的 RAM 阵列，如下图所示。

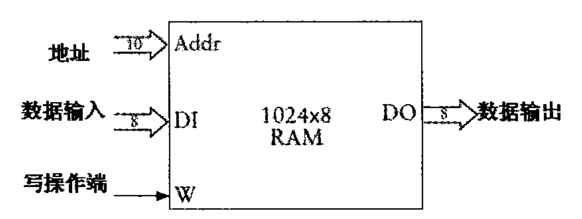


上图所示的 RAM 阵列存储容量为 16 个单位，每个单位占 1 位。

RAM 阵列的存储容量与其地址输入端的数量有直接的联系。在没有地址输入端的情况下（只有 1 位锁存器和 8 位锁存器的情况），只能存储 1 个单位的数据；当存在 1 个地址输入端时，可以存储 2 个单位的数据；有两个地址输入端时，可以存储 4 个单位的数据；有 3 个地址输入端时，可以存储 8 个单位的数据；有 4 个地址输入端时，可以存储 16 个单位的数据。我们可以把它们之间的关系归纳成如下等式：

RAM 阵列的存储容量 = 2 ^{地址输入端的个数}

前面已经向大家演示了怎么搭建小型 RAM 阵列，你可能会问：为什么不搭建一个大规模的 RAM 阵列呢？就像下面这样。



上图所示的 RAM 阵列可存储 8192 个比特的信息，每 8 个比特为一组，共分为 1024 个组。因为 2 的 10 次方恰好是 1024，所以地址端共有 10 个输入端口。电路还包括 8 位的数据输入端和 8 位的数据输出端。

从专业的角度来讲，这个 RAM 阵列的存储容量为 1024 个字节。就好比一个邮局放置了 1024 个邮箱，而每个邮箱里面都可以存放 1 字节大小的邮件（希望不是垃圾邮件）。

1024 字节通常简称为 1 千字节(kilobyte)，1K 这种称呼不可避免地要引起许多混淆。其中它的前缀 kilo（源于希腊文 khilioi，意思为 1000），经常在公制系统中用到。比如 1 千克(kilogram)代表着 1000 克(grams)；1 千米(kilometer)代表着 1000 米(meter)。有所不同的是，这里所说的 1 千字节却代表着 1024 个字节——并非 1000 个字节。

它们之间不同的根本原因在于公制系统是基于 10 的幂的计数系统，而计算机采用的是基于 2 的幂的计数系统，它们之间没有交集。比如 10 的幂为 10、100、1000、10000、100000 等，而 2 的幂为 2、4、8、16、32、64 等。我们可以证明不存在一对整数 a 和 b 使得 10 的 a 次幂与 2 的 b 次幂相等。

但是偶尔也会碰见非常接近的数字。事实的确如此，1000 十分接近 1024，用数学化的描述方法可以称这种关系为“约等于”，这样我们可以得到相应的数学表达式：

$$2^{10} \approx 10^3$$

这个表达式并非空穴来风，它真正的意义在于表明 2 的某次幂和 10 的某次幂几乎相等。我们利用这一巧合可以很方便地把 1024 个字节的存储空间用 1 千字节来表示。

千字节可以简写为 KB。这样我们可以说前面所讲过的那个 RAM 阵列存储能力为1024 个字节，也可以说成是 1KB。

绝不能认为 1KB 的 RAM 阵列的存储能力为 1000 字节，它实际上是大于 1000 字节，是 1024 个字节，为了准确而清晰地表达你脑海中的数据，我们可以使用“1 KB””或“1千字节”这两种通用的表述方式。

存储容量为 1KB 的存储系统由 8 个数据输入端、8 个数据输出端和 10 个地址输入端所组成。由于这些字节是由 10 个地址输入端来标识和访问的，所以这种 RAM 阵列存储容量为 2^{10} 个字节。如果我们再加上一条地址线，它的存储容量将变成原来的两倍。下面的公式表示了存储容量的翻倍的过程。

$$1\ \text{KB} = 1024\ \text{B} = 2^{10}\ \text{B} \approx 10^3\ \text{B}$$
$$2\ \text{KB} = 2048\ \text{B} = 2^{11}\ \text{B}$$
$$4\ \text{KB} = 4096\ \text{B} = 2^{12}\ \text{B}$$
$$8\ \text{KB} = 8192\ \text{B} = 2^{13}\ \text{B}$$
$$16\ \text{KB} = 16{,}384\ \text{B} = 2^{14}\ \text{B}$$
$$32\ \text{KB} = 32{,}768\ \text{B} = 2^{15}\ \text{B}$$
$$64\ \text{KB} = 65{,}536\ \text{B} = 2^{16}\ \text{B}$$
$$128\ \text{KB} = 131{,}072\ \text{B} = 2^{17}\ \text{B}$$
$$256\ \text{KB} = 262{,}144\ \text{B} = 2^{18}\ \text{B}$$
$$512\ \text{KB} = 524{,}288\ \text{B} = 2^{19}\ \text{B}$$
$$1{,}024\ \text{KB} = 1{,}048{,}576\ \text{B} = 2^{20}\ \text{B} \approx 10^6\ \text{B}$$

请注意最左侧一排的数字也以 2 的幂的顺序逐步递增。

我们把 1024 个字节简化成为了 1 KB，相同的逻辑，我们把 1024 KB 统称为 1 兆字节（megabyte，希腊文中的 mega 意味着宏大），兆字节通常缩写为 MB。下面这个例子表示了兆字节为单位的存储容量翻倍的过程。

$$1\ \text{MB} = 1{,}048{,}576\ \text{B} = 2^{20}\ \text{B} \approx 10^6\ \text{B}$$
$$2\ \text{MB} = 2{,}097{,}152\ \text{B} = 2^{21}\ \text{B}$$
$$4\ \text{MB} = 4{,}194{,}304\ \text{B} = 2^{22}\ \text{B}$$

$8\ MB = 8,388,608\ B = 2^{23}\ B$

$16\ MB = 16,777,216\ B = 2^{24}\ B$

$32\ MB = 33,554,432\ B = 2^{25}\ B$

$64\ MB = 67,108,864\ B = 2^{26}\ B$

$128\ MB = 134,217,728\ B = 2^{27}\ B$

$256\ MB = 268,435,456\ B = 2^{28}\ B$

$512\ MB = 536,870,912\ B = 2^{29}\ B$

$1,024\ MB = 1,073,741,824\ B = 2^{30}\ B \approx 10^{9}\ B$

希腊文中的 giga 意味着巨大，1024 MB 也就被顺其自然地称为 1 吉（gigabyte）字节，缩写为 GB。

同理，1 太字节（terabyte，teras 希腊语意思为巨人）表示 2^{40} 个字节（约为 10^{12}），也就是 1,099,511,627, 776 个字节，太字节的缩写为 TB。

1 KB 近似为 1000 个字节，1 MB 近似为 100 万个字节，1 GB 近似为 10 亿个字节，1 TB 近似为 1 万亿个字节。

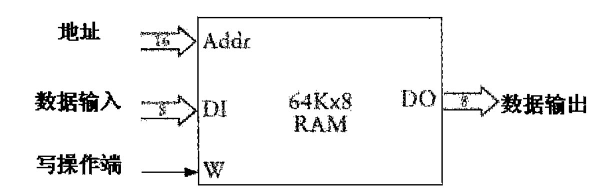
比 TB 还高数量级平时一般很少使用，比如 2^{50} 个字节表示为 1 批字节（petabyte），计算出来就是 1,125,899,906,842,624 个字节，约等于一千万亿，即 10^{15} 字节。1 安字节（exabyte）代表 2^{60} 个字节，也就是 1,152,921,504,606,846,976 个字节，约为 100 万的 3 次方，即 10^{18} 。

我们来补充一些生活中的基本常识。在写本书时（1999 年），家用电脑的随机存储器的容量一般为 32 MB、64 MB 或 128 MB（为了避免混淆，这里的任何描述都不涉及硬盘驱动器的任何内容，范围仅仅限定为 RAM），通过计算可以得到它们的存储大小分别为 33,554,432 个字节、67,108,864 个字节和 134,217,728 个字节。

简洁明了是我们人类交流方式的一大特色。比如，家中电脑内存大小如果为 65,536 字节，我们会说“我的是 64 K（这句话很可能是在 1980 年听到的）”；家中电脑内存大小如果为 33,554 ,432 字节，我们会说“我的是 32 M”。有少数电脑配备了 1,073,741,824 字节的内存，他们或许会说“我的可是上了 G 的（有时这句话的英文表述会让人误以为你在谈论音乐——I've got a gig）”。

有时人们可能会用到千比特或兆比特（注意是比特而不是字节），这只是极少数情况。当我们讨论涉及存储器的相关问题时，通常使用的是字节数而非比特（需要的时候可以通过把字节数乘以 8 将其转换成比特）。有一种情况下我们会经常用到千比特和兆比特，那就是在描述在线路中流动的数据时，很多句子中经常会出现千比特每秒（kbps）或兆比特每秒（mbps）这些用语。例如，一台 56K 的调制解调器指的是其数据处理速度为 56 千比特每秒，而不是 56 千字节每秒。

既然我们已经学会如何构造任意大小的 RAM 阵列，接下来继续对这个问题深究下去。假设现在已经构造好了一个容量为 65,536 字节的存储器组织，如下图所示。



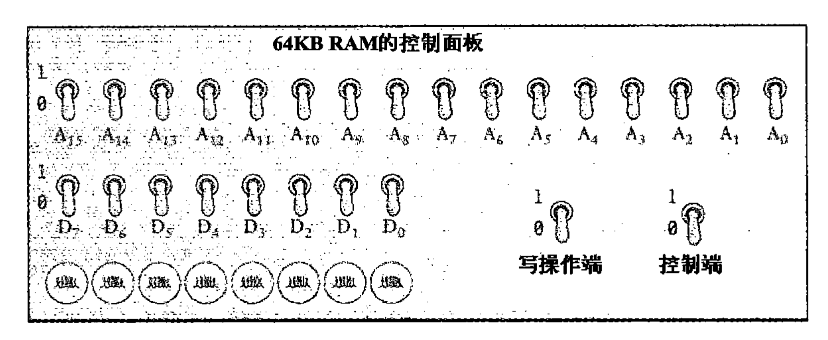
为什么选择大小为 64 KB 的 RAM 阵列？而非 32 KB 或 128 KB？因为 65,536 是一个约整数，转换为幂的形式就是 2^{16}，这个 RAM 阵列需要配备 16 位的寻址端。换句话说，该地址恰好可以用 2 个字节表示。将地址范围转化为十六进制就是 0000h～FFFFh。

我前面也提到过，64 KB 的内存是 1980 年的个人电脑的主流配置，但它的确不是用电报继电器组成的。我们可以用继电器来组成一块内存吗？我也相信你不想这么做。在我们先前的讨论中，存储每个比特需要 9 个继电器，推算一下 64K×8 的 RAM 阵列就需要至少 500 万个继电器！

如果用一种控制面板来辅助我们管理对这块 64KB 存储器的操作——包括写数据和读数据，一切将会直观明了。在这款控制面板上，有 16 个开关用于控制地址位，还有 8 个开关用来控制要输入的 8 比特数据。写操作端也用一个开关来表示，8 个灯泡用来显示 8 位数据，这个控制面板如下图所示。

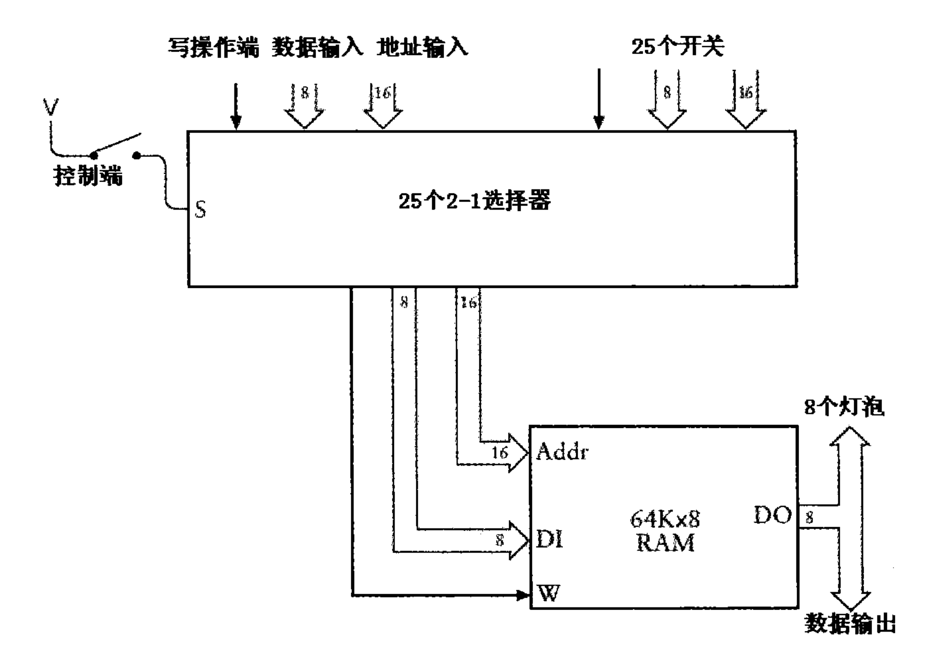
初始状态下所有的开关均置为 0。其中右下角有一个标识为控制端（takeover）的开关，这个开关的作用是确定由控制面板还是由外部所连接的其他电路来控制存储器。如果其他电路连接到与控制面板相连的存储器，这时控制端置 0（如图所示），此时存储器由其他电路系统接管，控制面板上的其他开关将不起任何作用；当控制端置 1 时，控制

面板将重新获得对存储器的控制能力。

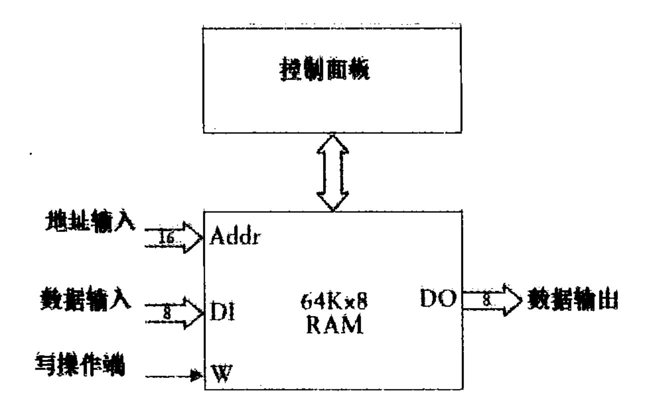


这种功能可以用一些 2-1 选择器来实现。仔细数一下会发现，我们需要 25 个 2-1 选择器——其中包括 16 个地址输入端、8 个数据输入端，以及 1 个写操作端。电路如下图所示。

当控制端开关断开时，RAM 阵列的地址端、数据输入和写操作端的数据全部来源于外部信号，也就是在 2-1 选择器的左上角的输入信号；当控制端开关闭合，RAM 阵列的地址端、数据输入端和写操作端的数据来源于控制面板开关发出的信号。但最终 RAM 阵列的输出信号都会传输到 8 个灯泡上或其他可能的地方。



下面这幅是控制面板与 64K×8 RAM 阵列的逻辑结构框图。



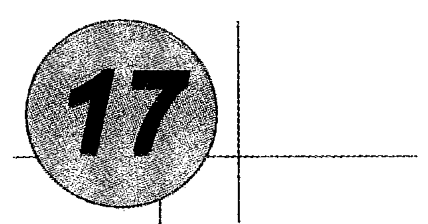
当控制端开关闭合时，通过操作 16 个地址开关，可以选择 65,536 个地址中的任何一个，灯泡的状态将表示该地址中所保存的 8 位数据。我们可以使用 8 个数据开关表示出一个新数，然后把写操作端置 1，从而将数据写入存储器。

64K×8 的 RAM 阵列和控制面板这一组合的确很实用，它可以帮助我们存储 65,536 个 8 位数据并且读取其中的任意一个。与此同时，我们也给其他部件提供了接入系统的机会——需要接入系统的通常是一些电路部件——这些部件可以轻易地读取并利用存储器中存放的数据，还可以把数据写入存储器。

关于存储器有一个问题尤其值得我们注意，而且需要特别注意。在学习第 11 章的时候，我们曾介绍过逻辑门的概念及原理，但是没有画出组成逻辑门的单个继电器的结构图。特别是，当时没有指明每个继电器都与某个电源连接在一起。只要继电器连通，电流就会流过电磁线圈并产生磁场，继而吸下金属片。

一个辛辛苦苦装满 65,536 字节珍贵数据的 64 K×8 RAM 阵列，如果断掉电源，会发生什么事情呢？首先所有的电磁铁都将因为没有电流而失去磁性，随着"梆"的一声，金属片将弹回原位，所有继电器将还原到未触发状态。RAM 中存储的数据呢？它们将如风中残烛般消失在黑暗中。

正因为如此，随机访问存储器也被称为易失性（volatile）存储器。为了保证存储的数据不丢失，易失性存储器需要恒定的电流。



自动操作

我们人类的创造能力与勤奋精神常常令我感叹不已，但人类的本性却是相当懒惰的。举个简单而又常见的例子，我们总是不情愿工作。我们对工作的反感是如此的强烈——当然人类也很聪明——以至于情愿花费大量的时间去设计并制造一些设备，哪怕这些设备只能将工作时间缩减几分钟。悠闲地躺在吊床上，看着自己刚发明的新奇工具自动修剪草坪，没有什么事情能比这更让我们快乐的神经为之一动了。

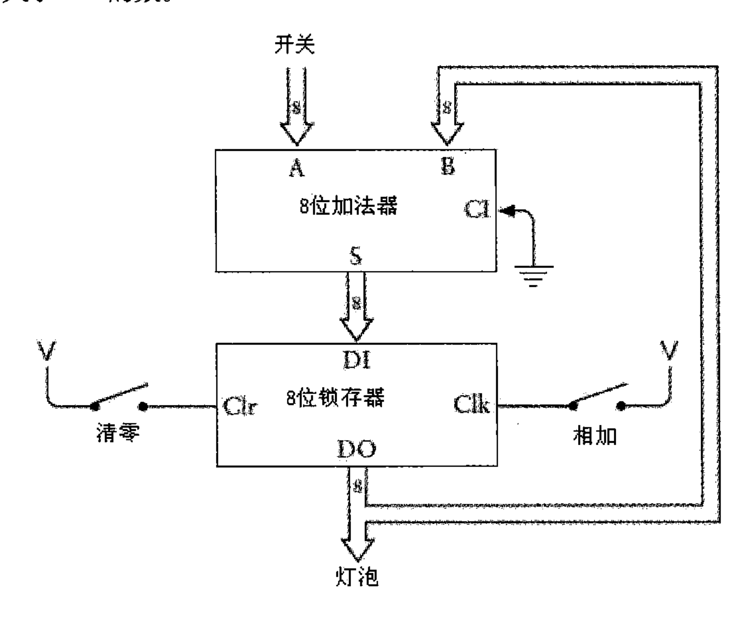
很可惜，我并不打算在本章中介绍自动割草机的设计。在这里我们将学会设计更加先进的机器，目标就是要使加减法的过程自动化，这听起来有些难以置信。但是，本章最终设计出的机器用途将十分广泛，因为它可以解决所有能用加、减法处理的问题，而事实上现实中的很多问题都是可以用加法与减法来解决的。

当然，随着机器变得越来越精密，其复杂程度也越来越高，因此对其构造的某些部分理解起来比较困难。因此如果你略去了某些复杂的细节，这也是在情理之中，没有人会为此责怪你。有时，我们会感到焦躁不安，甚至厌倦，发誓再也不会为了解决一个复杂的数学难题而去求助于某些电子或机械设备。但是请保持耐心，在本章的末尾我们将会设计出一种机器，我们可以称它为计算机（Computer）。

回忆我们曾在第 14 章讨论过的一个加法器。这个版本的加法器包括一个 8 位的锁存

器，用于对 8 个开关的输入数据进行迭代求和。下面是其结构图。

从图中可以看出，8 位锁存器利用触发器来保存 8 位数据。使用这个设备时，首先需要按下清零开关使锁存器中的内容全部都变为 0，然后通过开关输入第一个数。加法器只是简单地将这个数字和锁存器输出的 0 进行求和，因此相加的结果与原先输入的数字是一样的。按下相加开关可以把这个数保存在锁存器中，最后会点亮某些灯泡以显示它。现在通过开关输入第二个数，加法器把它与已经存放在锁存器中的第一个数相加。再次按下相加开关，就可以把相加的结果存入锁存器中，并通过灯泡显示这个结果。通过这种方式，可以把一串数相加并显示运行结果。显然，这种设计方案存在一个缺陷：8 个灯泡无法显示大于 255 的数。



对于第 14 章所介绍的这种电路，目前为止只讲到了一种锁存器，它是电平触发（level triggered）的。在电平触发的锁存器中，为了保存数据必须将时钟输入端首先置 1，然后回置为 0。当时钟输入端为 1 时，锁存器的数据输入端可以改变，而这些变化将会影响到数据输出。在第 14 章的后半部分还介绍了边沿触发（edge-triggered）的锁存器，这种锁存器在时钟输入从 0 跳变为 1 的瞬间保存数据。边沿触发器在很多方面更加易于使用，因此假定本章用到的所有触发器都是边沿触发的。

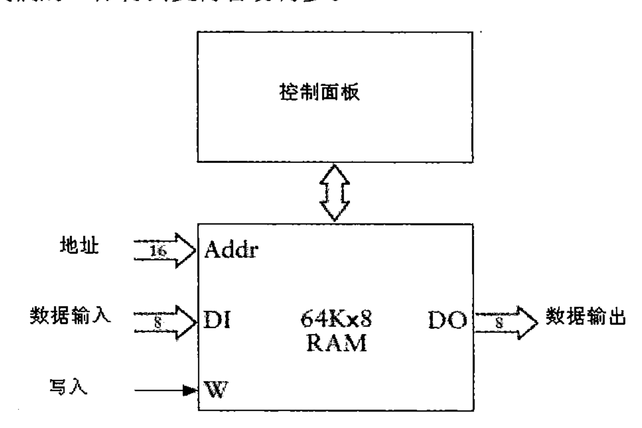
用来累加多个数的锁存器称做累加器（accumulator）。在本章的后面将会看到累加器

不仅仅做简单的累加，它还充当着锁存器的角色，保存第一个数，并且和下一个数做加法或减法运算。

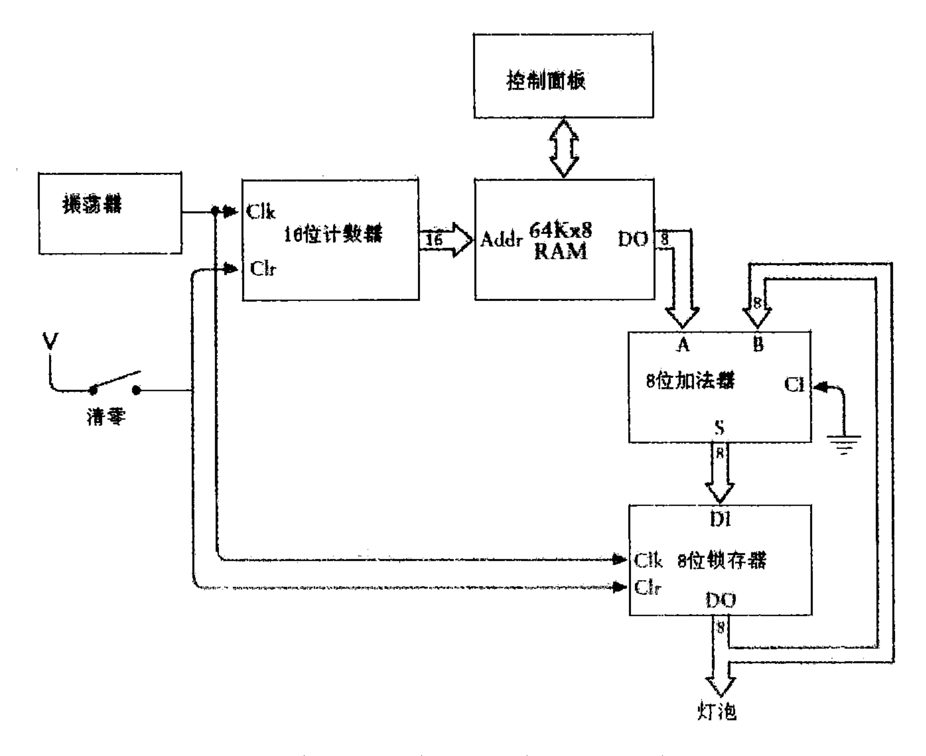
很显然，上面的加法器存在着一个很大的缺陷：假如要把 100 个二进制数加起来，你必须端坐于加法器前，并且耐心地输入所有的数并累加起来。但是当你终于完成时，却发现其中有两个数输错了，而你只能重复一遍所有的工作。

但是，也许并非如此。在前一章我们使用了大约 500 万个继电器构造了一个 64 KB 的 RAM 阵列。除此之外，我们还把一个控制面板连接到电路帮助我们工作，闭合它的控制（Takeover，有些书中也称“接管”）端开关后，就可以使用其他开关来控制 RAM 阵列的读写。下面是 64 KB RAM 阵列结构图。

如果把这 100 个二进制数输入到 RAM 阵列中而不是直接输入到加法器中，一旦需要修改一些数据，我们的工作将会变得容易得多。



因此我们所现在面临的挑战就是如何把 RAM 阵列和累加器连接起来。很显然，RAM 阵列的输出信号可以替代加法器的开关。而你也许想不到，用一个 16 位的计数器（比如我们在 14 章构造的那种）就可以控制 RAM 阵列的地址信号。在这个电路中，RAM 阵列的数据输入信号和写操作端信号可以省去。修改后的电路结构如下图所示。



当然，这并不是迄今发明的最易于使用的计算设备。要使用它，首先要闭合清零开关，这样做的目的是，清除锁存器中的内容并把 16 位计数器的输出置为 0000h。然后闭合 RAM 控制面板的控制端开关。现在你可以从地址 0000h 开始输入一组你想要相加的 8 位数。如果有 100 个数，那么它们将被存放在 0000h ~ 0063h 的地址空间中(也应该把 RAM 阵列中未使用的单元设置为 00h)。然后闭合 RAM 控制面板的控制端开关（这样控制面板就不再控制 RAM 阵列了），同时断开清零开关。做完了这些，我们可以静静地坐下来，观察灯泡显示运算结果。

让我们来看一下它是怎样工作的：当清零开关第一次断开时，RAM 阵列的地址输入是 0000h。RAM 阵列的该地址中存放的 8 位数值是加法器的输入数据。加法器的另一个输入数据为 00h，因为此时锁存器也已经清零了振荡器提供的时钟信号———个可以在 0，1 之间快速切换的信号。清零开关断开后，当时钟信号由 0 跳变为 1 时，将有两件事同时发生：锁存器保存加法器的计算结果，同时 16 位计数器增 1，指向 RAM 阵列的下一个地址单元。清零开关断开之后，时钟信号第一次从 0 跳变为 1 时，锁存器就将第一个数值保存下来，同时计数器增加为 0001h；当时钟发生第二次跳变时，锁存器保存之前两个数的求和结果，同时计数器增加

为0002h；按这种方式往复操作。

要注意的是，这里首先做了一些假设。最主要的一点就是，振荡器要足够慢以使电路的其他部分可以工作。每次时钟振荡的过程中，在加法器输出有效的结果之前，一些继电器必须去触发其他继电器。

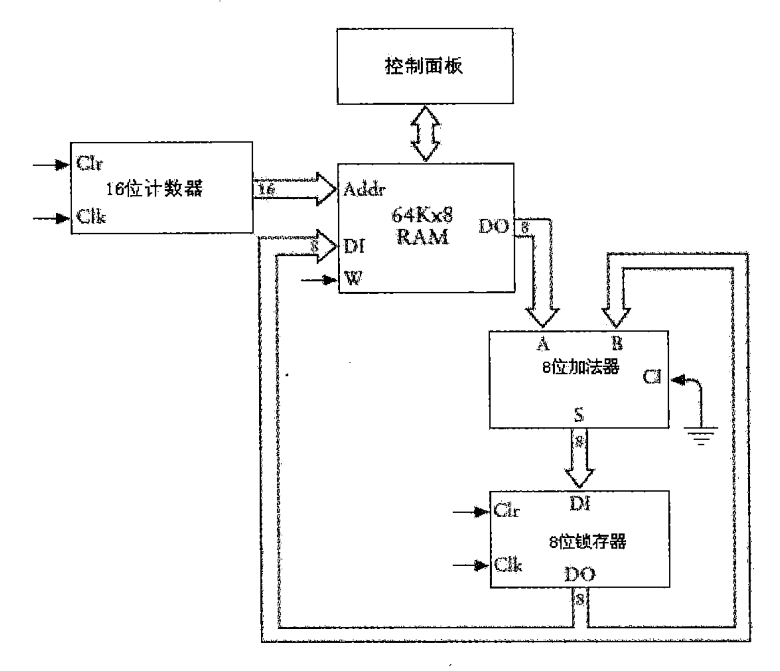
这个电路存在的一个缺陷是：我们没有办法使它停下来。在某一个时刻，所有灯泡会停止发光，因为RAM阵列的剩余部分存放的数都是00h。这时，你可以读取二进制的运算结果。但是当计数器达到FFFFh时，它会重新回滚（roll over）到0000h（这就好像汽车的里程表一样），这时自动加法器会再一次把所有的数累加到已经计算出来的结果中去。

这个加法器还存在另一个问题：它只能做加法运算，并且只能做8位数的加法。在这个RAM阵列中，不但每一个数要小于255，而且任意个数相加的结果也要小于255。此外，该加法器也不能处理减法运算，尽管可以用2的补数表示负数，但在这种情况下加法器能处理的数字的范围被限制在-128到127之间。要处理更大的数（例如，16位数）的话，一个简单的方法是：把RAM阵列、加法器、锁存器的位宽全都加倍，同时增加8个灯泡。但这些投资在我们看来是不合算的。

当然，这里提到这个问题的原因是最终我们要解决它。但首先来关注另一个问题。如果你不需要把100个数加在一起呢？如果你想做的是用自动加法器把50对数分别相加，得出50个不同的结果呢？或者你需要一种万能机，它可以方便地对两个数，10个数甚至100个数求和，并且所有的计算结果都可以很方便地使用。

先前提到的自动加法器都是用连接在锁存器上的灯泡来显示运行结果的，但是如果你想对50对数分别求和的时候，这就不是一个好的方法了。你可能会想到把运算结果存回到RAM阵列中去，这样的话，就可以在适当的时候用RAM阵列的控制面板来检查运算结果。为了实现这个目的，控制面板上专门设计了灯泡。

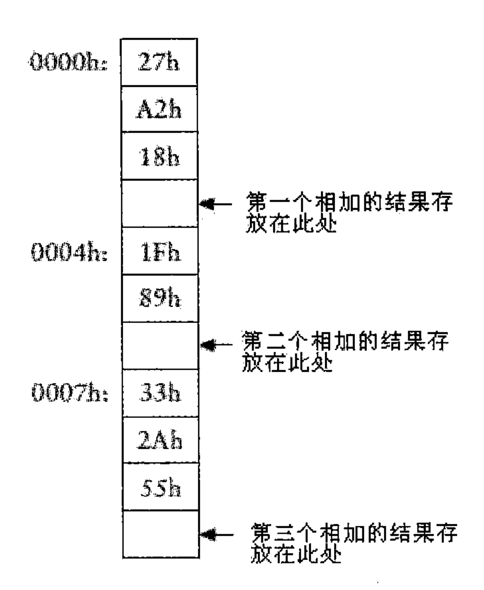
这意味着我们可以去掉与锁存器连接的灯泡，取而代之的是把锁存器的输出端连接到RAM阵列的数据输入端，这样就可以把计算结果写回到RAM阵列中去，如下图所示。



上图中略去了自动加法器的其他部分，其中包括振荡器和清零开关，这是因为我们不再需要特别标注计数器和锁存器的清零及时钟输入。此外，既然我们现在已经开始利用 RAM 的数据输入，因此需要一种用来控制 RAM 写入信号的方法。

现在我们不需要担心电路能否工作，而要把注意力集中到急需解决的问题上来。目前的当务之急是如何配置一个自动加法器，使它不仅仅可以对一组数字做累加运算，还希望它能够自主地确定要累加多少个数字，而且还能记住在 RAM 中存放了多少个计算结果，这样就可以简化查询工作。

例如，假设我们先要对三个数进行求和，然后对两个数进行求和，最后再对三个数进行求和。想象一下，我们可以把这些数保存在 RAM 阵列中以 0000h 开始的一组空间中，这些数存储在 RAM 阵列中的具体形式如下图所示。



本书中将用这样的形式表示一小段存储器。方格表示的是存储器的内容。存储器的每一个字节写在一个方格里。地址标记在方格的左边，并不是每一个地址都需要标记，因为地址是线性的，所以总是可以通过计算确定某个方格对应的地址。方格右边是关于该存储单元内容的注释，这些标记的单元就是我们想要自动加法器保存三个计算结果的位置（尽管这些方格画出来是空的，但是存储单元内并不是空的，它们总是保存着一些东西，就算只是一些随机数，但此时存放的是一些没有用的数）。

或许大家都有一种冲动，想亲自去做十六进制计算，并把结果填到那些小格子中去，但这并不是实验的目的，我们想要自动加法器为我们做这些加法。

我们并不希望自动加法器成为单任务系统——在它的第一个版本中，只是把 RAM 地址中的内容加到称为累加器的 8 位锁存器中——实际上我们希望它能做四件事：进行加法操作，首先它要把一个字节从存储器中传送到累加器中，这个操作称为加载（Load）。第二个操作把存储器中的一个字节加（Add）到累加器的内容中去。第三个操作把累加器中的计算结果取出并存放到存储器中。另外我们需要用一个方法令自动加法器停（Halt）下来。

我们借助具体的例子详细介绍这一过程，以上文提到的自动加法器所做的运算为例来说明。

（1）把 0000h 地址处的内容加载到累加器。

（2）把 0001h 地址处的内容加到累加器中。

（3）把 0002h 地址处的内容加到累加器中。

（4）把累加器中的内容存储到 0003h 地址处。

（5）把 0004h 地址处的内容加载到累加器。

（6）把 0005h 地址处的内容加到累加器。

（7）把累加器中的内容存储到 0006h 地址处。

（8）把 0007h 地址处的内容加载到累加器。

（9）把 0008h 地址处的内容加到累加器。

（10）把 0009h 地址处的内容加到累加器。

（11）把累加器中的内容存储到 000Ah 地址处。

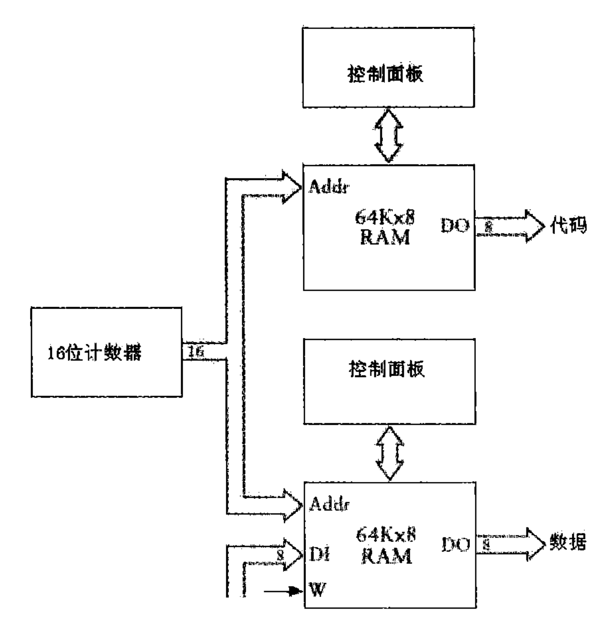
（12）令自动加法器停止工作。

这里要注意，同最初的自动加法器一样，存储器中的每一个字节的地址仍然是以 0000h 为起点线性排列的。最初的加法器只是简单地把存储器指定地址的内容和累加器中的内容相加，在某些情况下需要这样做。但是有时我们需要把存储器中的某个值直接加载到累加器，或者把累加器中的值直接保存到存储器。做完了这些工作，算得上万事俱备只欠东风了，我们还希望自动加法器能方便地停下来，以便于查看 RAM 阵列中存放的值。

该如何来完成这些工作呢？能不能仅仅简单地向 RAM 阵列中输入一组数，然后期待自动加法器正确地完成所有工作呢？答案是否定的。对于 RAM 阵列中的每一个数，我们还需要用一些数字代码来标识加法器要做的每一项工作：加载、相加、保存和终止。

也许存放这些代码的最简单的方法（但肯定不是代价最小的）是把它们存放在一个独立的 RAM 阵列中。这个 RAM 应该和第一个 RAM 同时被访问。但是这个 RAM 中存放的是不需要求和的数，而是一些数字代码，用来标记自动加法器对第一个 RAM 中指定地址要做的一种操作。这两个 RAM 可以分别被标记为“数据”（第一个 RAM 阵列）和

“代码”（第二个 RAM 阵列）。其结构如下图所示。



我们已经清楚地认识到新的自动加法器能够把数据求和的结果写入到第一个 RAM 阵列（标记为“数据”），而新的 RAM 阵列（标记为“代码”）则只能通过控制面板写入。

我们需要四个代码来标记新的自动加法器需要做的四个操作，这些代码可以任意指定。如下所示的是一种方案。

操作码	代码
Load（加载）	10h
Store（保存）	11h
Add（加法）	20h
Halt（停止）	FFh

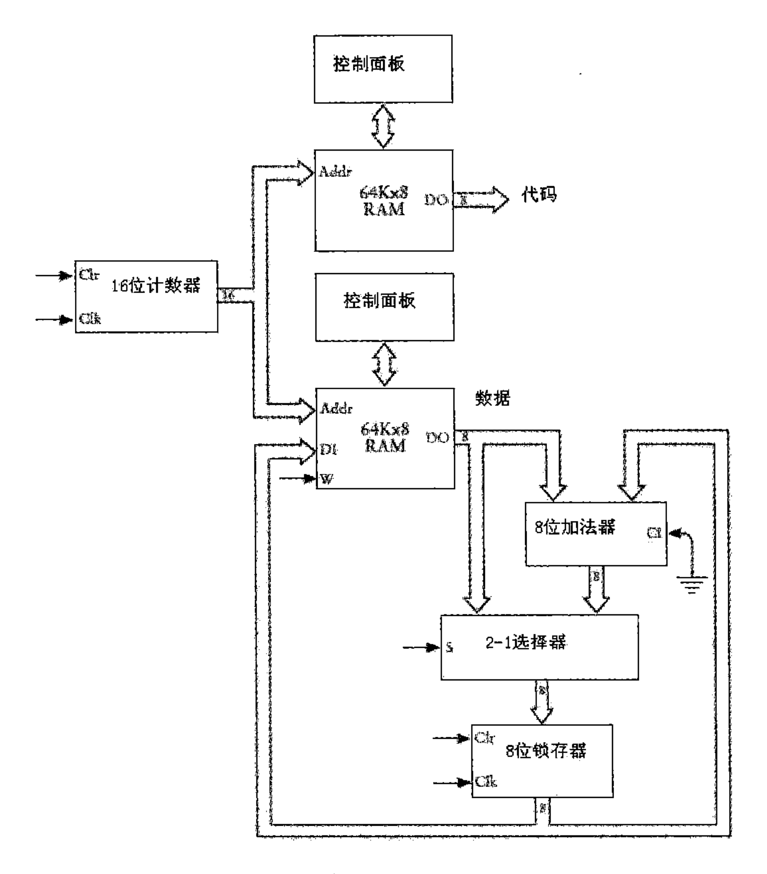
为了使上面讨论的三组加法得以正常执行，你需要通过控制面板把如下值存入代码 RAM 阵列。

比较一下该 RAM 阵列与存放累加数据的 RAM 阵列中的内容，你会发现，代码 RAM 阵列中存放的每一个代码都对应着数据 RAM 中要被加载或者加到累加器中的数，或者对应需要存回到数据 RAM 中的某个数。以这种方式使用的数字代码常常被称为指令码（instruction code）或操作码（operation code，opcode）。它们指示电路要执行的某种操作。

地址	内容	指令
0000h:	10h	Load
	20h	Add
	20h	Add
	11h	Store
0004h:	10h	Load
	20h	Add
	11h	Store
0007h:	10h	Load
	20h	Add
	20h	Add
	11h	Store
000Bh:	FFh	Halt

如前所述，最初的自动加法器的 8 位锁存器的输出要作为数据 RAM 阵列的输入，这就是 Save 指令的功能。还需要做另一个改变：以前 8 位加法器的输出是 8 位锁存器的输入，但现在为了执行 Load 指令，数据 RAM 阵列的输出有时也要作为 8 位锁存器的输入，这种新的变化需要一个 2-1 选择器来实现。改进后的自动加法器如下图所示。

图中略去了一些组件，但是仍然清晰地描述了各个组件之间的 8 位数据通路。16 位的计数器为两个 RAM 阵列提供地址输入。通常，数据 RAM 阵列的输出传入到 8 位加法器执行加操作。8 位锁存器的输入可以是数据 RAM 阵列的输出（当执行 Load 指令时），也可以是加法器的输出（当执行 Add 指令时），这种情况下就需要 2-1 选择器。通常，锁存器电路的输出又流回到加法器中，但是当执行 Save 指令时，它就成为了数据 RAM 阵列的输入数据。

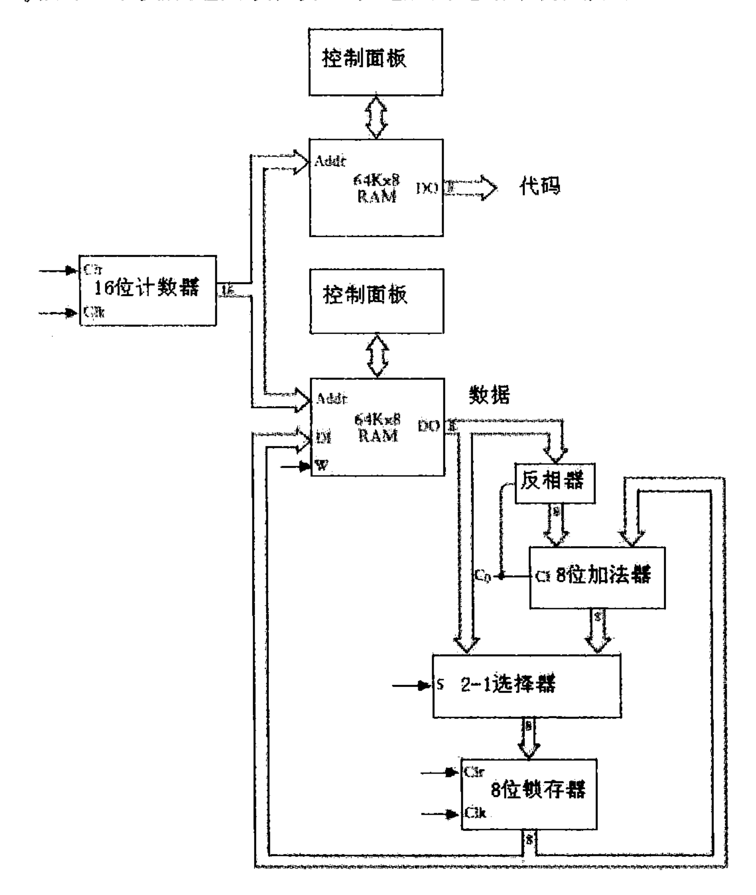


上图缺少的是控制所有这些组件的信号，它们统称为控制信号，包括16位计数器的“时钟”输入和“清零”输入，8位锁存器的“时钟”输入和“清零”输入，数据RAM阵列的“写”（W）输入，2-1 选择器的“选择”（S）输入。其中的一些信号很明显是基于代码RAM阵列的输出，例如，如果代码RAM阵列输出是Load指令，那么2-1选择器的“选择”输入必须是0（即选择数据RAM的输出）。只有当操作码是指令Store时，数据RAM阵列的“写”（W）输入必须是1。这些控制信号可以通过逻辑门的各种组合来实现。

利用最少的附加硬件和一些新增的操作码，可以让这个电路从累加器中减去一个数。第1步是向操作码表增加一些代码。

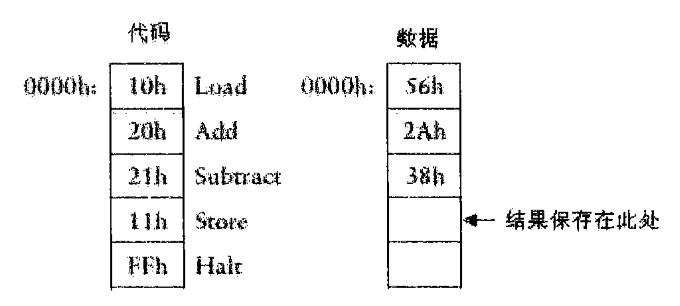
操作码	代码
Load	10h
Store	11h
Add	20h
Subtract（减法）	21h
Halt	FFh

对于 Add 和 Subtract 的代码，其区别仅在于最低有效位，我们称该位为 C_0。如果操作码为 21h，除了数据 RAM 阵列的数据传入加法器之前要取反，并且加法器进位输入置 1 之外，电路所做的操作与执行 Add 指令所做的操作相同。在这个增加了一个反相器的改进电路中，C_0 信号可以完成这两项任务。改进后的电路结构图如下。



假设现在要把 56h 和 2Ah 相加，然后再从中减去 38h，可以按照下图中两个 RAM 阵

列中的代码（操作码）和数据（操作数）完成该运算。



Load 操作完成之后，累加器中的值更新为 56h，加法操作完成后累加器中的值为 56h 与 2Ah 的和，即 80h。Subtract 操作使数据 RAM 阵列的下一个值（38h）按位取反，得到 C7h。当加法器的进位输入置 1 时，取反得到 C7h，然后使其与 80h 相加：

$$\begin{array}{r} \text{C7h} \\ +\ \text{80h} \\ +\ \ \text{1h} \\ \hline \text{48h} \end{array}$$

最后的结果是 48h。（在十进制中，86 加 42 再减去 56 等于 72）

还有一个一直没有找到合适的解决办法的问题：加法器及连接到它的所有设备的宽度只有 8 位。以前提出过的一个解决办法是把两个 8 位加法器（其他的大部分设备也用两个）连在一起，构成一个 16 位的设备。

但还有代价更小的解决办法，假如你想把两个 16 位的数相加，比如：

$$\begin{array}{r} \text{76ABh} \\ +\ \text{232Ch} \\ \hline \end{array}$$

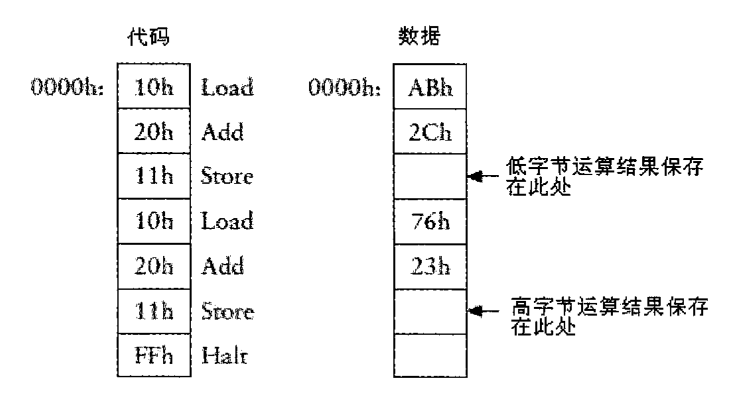
这种 16 位的加法先单独处理最右边的字节（通常称之为低字节）：

$$\begin{array}{r} \text{ABh} \\ +\text{2Ch} \\ \hline \text{D7h} \end{array}$$

然后再计算最左边的字节，即高字节的和：

$$\begin{array}{r} 76h \\ +23h \\ \hline 99h \end{array}$$

得到相同的结果 99D7h。因此，如果我们把两个 16 位的数用这种方式保存在存储器中，就像下面这样：



运算结果 D7h 将被保存到地址 0002h，而结果 99h 将被保存到地址 0005h。

当然并非所有的情况都是这样处理，只是上面的例子中用到了这种方法。如果要把 76ABh 和 236Ch 这两个 16 位的数相加该怎么做呢？在这个例子中，对两个数的低字节求和时将会产生一个进位：

$$\begin{array}{r} ABh \\ +6Ch \\ \hline 117h \end{array}$$

产生的这个进位必须与两个数的高字节的和再相加：

$$\begin{array}{r} 1h \\ +76h \\ +23h \\ \hline 9Ah \end{array}$$

最后的计算结果为 9A17h。

我们能够改进自动加法器的电路，使它可以正确地进行 16 位数的加法操作吗？答案是肯定的，我们需要做的仅仅是在第一步运算时保存低字节数运算的进位输出，并把它

作为下一步高字节数运算的进位输入。如何保存1位呢？1位锁存器就是最好的选择了，该锁存器应该被称为进位锁存器（Carry latch）。

为了使用进位锁存器，还需要另一个操作码，我们称之为“进位加法”（Add with Carry）。当进行8位数加法时，使用的是常规的Add指令。加法器的进位输入是0，它的进位输出将会保存到进位锁存器（尽管它根本不会被用到）。

如果要对两个16位的数进行加法运算，我们仍然使用常规的Add指令对两个低字节数进行加法运算。加法器的进位输入是0，而其进位输出被锁存到进位锁存器中。当把两个高字节数相加时，要使用新的Add with Carry指令。在这种情况下，两个数相加时要用进位锁存器的输出作为加法器的进位输入。因此，如果第一步低字节数的加法运算有进位，则该进位将用于第二步高字节数的加法运算；如果没有进位，则进位锁存器的输出是0。

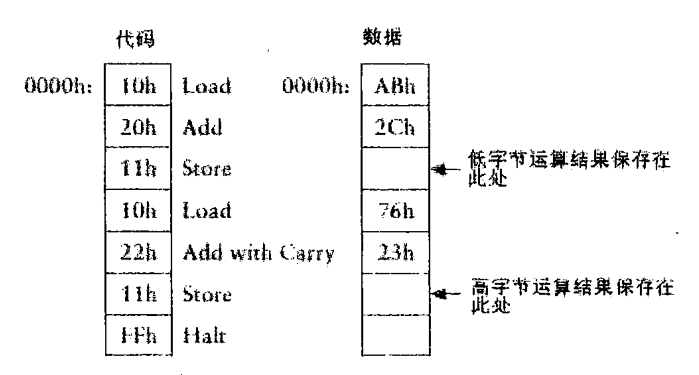
如果要进行16位数的减法运算，则还需要一个新的指令，称为“借位减法”（Subtract and Borrow）。通常，Subtract指令需要将减数取反并且把加法器的进位输入置1。进位输出通常不是1，因此应该被忽略。但对16位数进行减法运算时，进位输出应该保存在进位锁存器中。在进行第二步的高字节减法运算时，锁存器保存的结果应该作为加法器的进位输入。

在加入了Add with Carry和Subtract and Borrow之后，目前我们已经有了7个操作码，如下表所示。

操作码	代码
Load	10h
Store	11h
Add	20h
Subtract	21h
Add with Carry（进位加法）	22h
Subtract with Borrow（借位减法）	23h
Halt	FFh

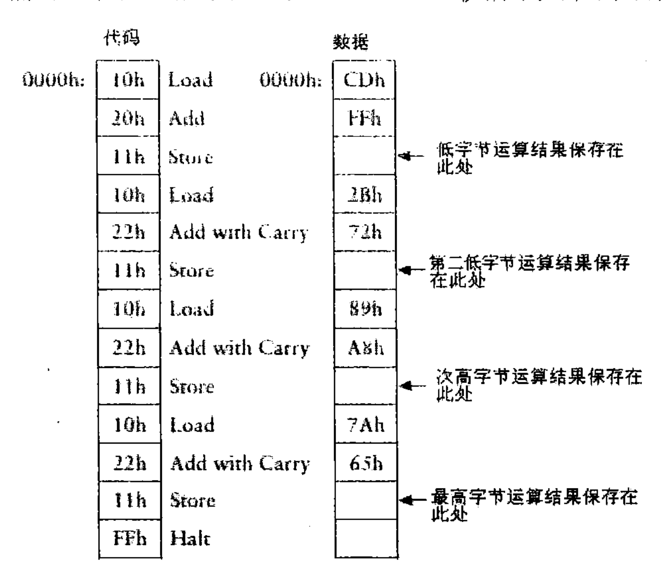
在执行减法或借位减法运算时，送入加法器中的操作数需要进行取反预处理。加法器的进位输出是进位锁存器的数据输入。无论何时，执行加法、减法、进位加法或借位减法中的任一种运算，进位锁存器都是同步的。当执行减法运算，或进位锁存器的数据输入为1且正在执行进位加法或者借位减法运算时，8位加法器的进位输入都是置1的。

需要记住的是，只有当前一次的加法或者进位加法操作使加法器产生进位输出时，Add with Carry 指令才会使 8 位加法器的进位输入置 1。因此，只要进行多字节数加法运算，不管实际是否需要，都应该使用 Add with Carry 指令。为了保证编码的正确，使前面提到 16 位加法正常进行，可用如下方法。



不论操作数是什么，该方法都可以正确执行。

增加了两个新的操作码之后，我们已经极大地扩展了加法器的功能，它不再局限于 8 位数的加法运算。通过执行进位加法操作，可以对 16 位数、24 位数、32 位数、40 位数，甚至更多位的数进行加法运算。假如要进行两个 32 位数 7A892BCDh 和 65A872FFh 的加法运算，我们仅需要 1 条 Add 指令和 3 条 Add with Carry 指令，如下图所示。



当然，把这些数依次输入存储器并不是最好的做法。因为你不但要使用开关来输入这些数，而且保存这些数的存储单元的地址也不是连续的。例如，7A892BCDh 从最低字节开始，每个字节依次保存在 0000h，0003h，0006h，0009h 中。而为了得到最后的结果，还需要检查 0002h，0005h，0008h，000Bh 这几个地址中的数。

除此之外，当前设计的自动加法器不允许在随后的计算中重复使用前面的计算结果。假设我们要对三个 8 位数求和，然后再从中减去一个 8 位数并保存结果。这可能需要一条 Load 指令，两条 Add 指令，一条 Subtract 指令以及一条 Store 指令。但如果想从原来的求和结果（3 个 8 位数的和）中减去另一个数该怎么做呢？这个求和结果已经不能被访问了，每次我们使用它的时候都必须重新计算。

产生上述情况的原因就在于我们构造的自动加法器具有如下的特性：它的代码存储器和数据存储器是同步的、顺序的，并且都从 0000h 开始寻址。代码存储器中的每一条指令对应数据存储器中相同地址的存储单元。一旦执行了一条 Store 指令，相应的，就会有一个数被保存到数据存储器中，而这个数将不能重新加载到累加器。

要解决这个难题，需要对自动加法器的设计做一个根本性的且程度极大的修改。这个想法实现起来似乎非常困难，但是很快你就会发现（我希望是这样）改进后的加法器具有更高的灵活性。

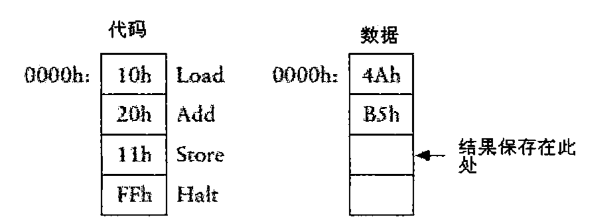
现在让我们立刻开始吧，目前已经有了 7 个操作码，如下所示。

操作码	代码
Load	10h
Store	11h
Add	20h
Subtract	21h
Add with Carry（进位加法）	22h
Subtrac t with Borrow（借位减法）	23h
Halt	FFh

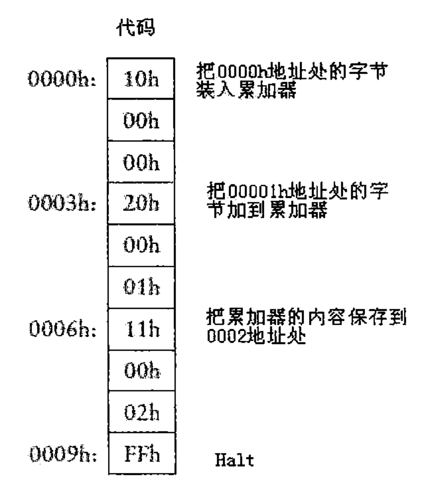
每一个操作码在存储器中占 1 个字节。现在除了 Halt 操作码外，我希望每一个指令在存储器中仅占据 3 个字节的空间，其中第一个字节为代码本身，另外的两个字节用来存放 1 个 16 位存储器单元地址。对于 Load 指令来说，后两个字节保存的地址用来指明数据 RAM 阵列的一个存储单元，该单元存放的是需要被加载到累加器中的字节。对于

Add，Subtract，Add with Carry，Subtract with Borrow 指令来说，该地址指明的存储单元所保存的是要从累加器中加上或减去的字节。对于 Store 指令来说，该地址指明的是累加器中的内容将要保存到的存储单元地址。

例如，当前加法器所能进行的最简单的运算就是对两个数求和。为了执行这个操作，需要按下面的方式设置代码 RAM 阵列和数据 RAM 阵列。



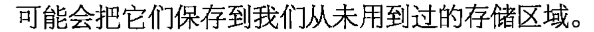
在改进的自动加法器中，每条指令（除了 Halt 指令）需要 3 个字节。

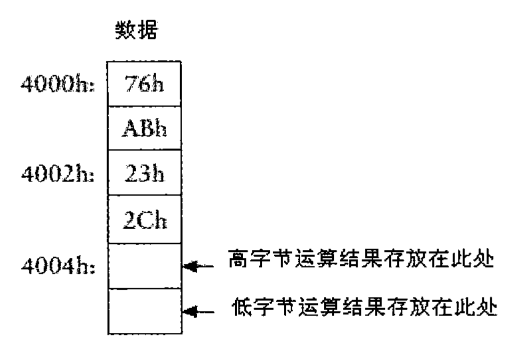


每一条指令的代码（除了 Halt 指令）后跟两个字节，用来指明数据 RAM 阵列中 16 位的存储地址。这三个地址恰巧是 0000h，0001h 和 0002h，但它们可以是任何其他可用的地址。

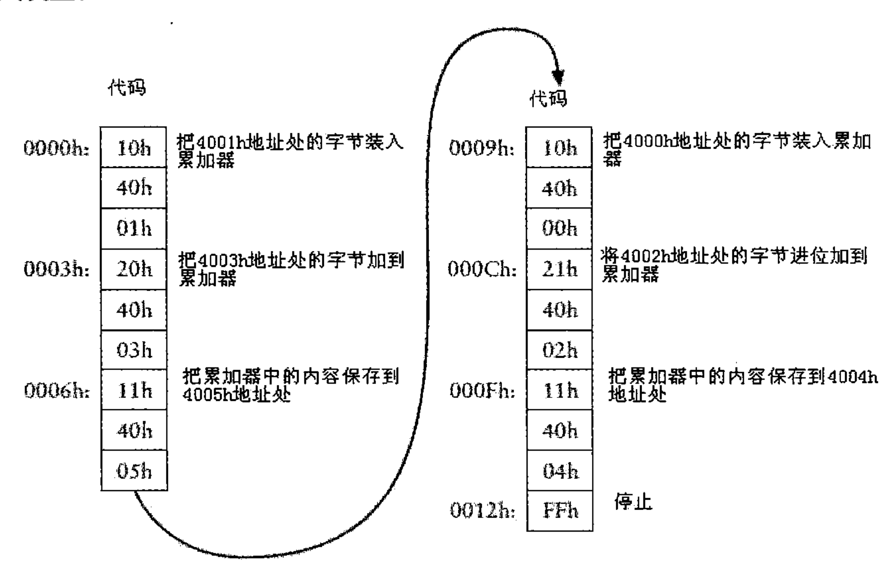
前面讲到了如何用 Add 或 Add with Carry 指令来对两个 16 位数——比如 76ABh 和 232Ch 求和。必须把两个数的低字节保存到存储器的 0000h 和 0001h 地址，把其高字节保存到 0003h 和 0004h 地址，运算的结果分别保存在 0002h 和 0005h。

通过这种变化，我们可以用一种更合理的方式来保存这两个操作数及其运算结果，


可能会把它们保存到我们从未用到过的存储区域。



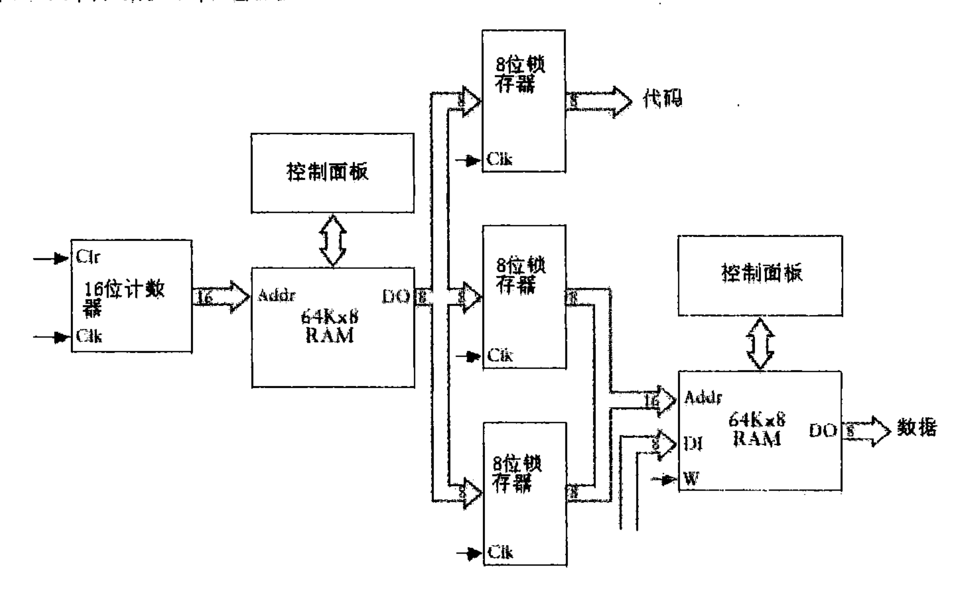
这 6 个存储单元不必像上图中这样全都连在一起，它们可以分散在整个 64 KB 数据 RAM 阵列的任意位置。为了把这些地址中的数相加，代码 RAM 阵列中的指令必须用以下方式设置。



可以看到，保存在地址 4001h 和 4003h 处的两个低字节数先执行加法，其结果保存在 4005h 地址处。两个高字节数（分别保存在 4000h 和 4002h 处）通过 Add with Carry 指令相加，其结果保存在地址 4004h 处。如果去掉 Halt 指令并向代码 RAM 中加入更多指令，随后的计算可以通过引用地址很方便地使用原来的那些操作数及其结果。

实现该设计的关键是把代码 RAM 阵列的数据输出到 3 个 8 位锁存器中。每个锁存器

保存该 3 字节指令的一个字节。第一个锁存器保存指令代码本身，第二个锁存器保存地址的高字节，第三个锁存器保存地址的低字节。第二个和第三个锁存器的输出构成了数据 RAM 阵列的 16 位地址。



从存储器中取出指令的过程称为取指令（instruction fetch）。在我们设计的加法器中，每一条指令的长度是 3 个字节。因为每次从存储器取回一个字节，所以取每条指令需要的时间为 3 个时钟周期。此外，一个完整的指令周期需要 4 个时钟周期。这些变化必然使得控制信号更加复杂。

机器响应指令码做一系列操作的过程称为执行（execute）指令，但这并不能表明机器是一种有生命的东西，因为它不能自行分析机器代码并决定该做什么。每一种机器码用其唯一的方式触发多种控制信号，从而引发机器执行各种操作。

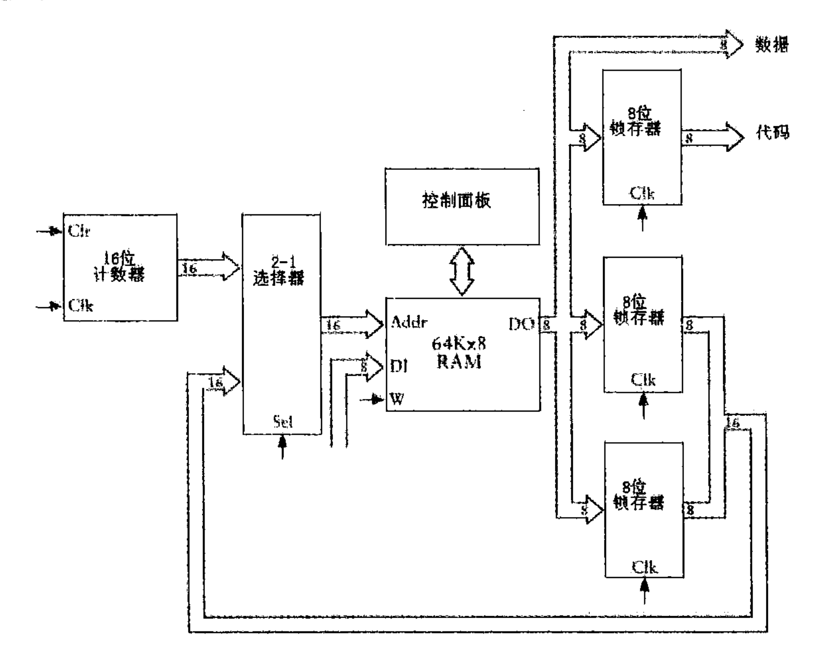
注意，为了让上面的加法器更加有用，我们牺牲了运算速度。使用同样的振荡器，它的运算速度只有本章提到的第一个加法器的 1/4。这验证了一个称为 TANSTAAFL 的工程准则，它的意思是“天下没有免费的午餐”。通常上帝总是很公平的，你改进了机器的某个方面，则其他方面就会受到损失，有得就有失。

如果不使用继电器构造这个电路的话，很显然，两个 64 KB 的 RAM 阵列构成了电路中最主要的部分。事实上早就应该放弃这些部件了，甚至从一开始就应该决定只需要

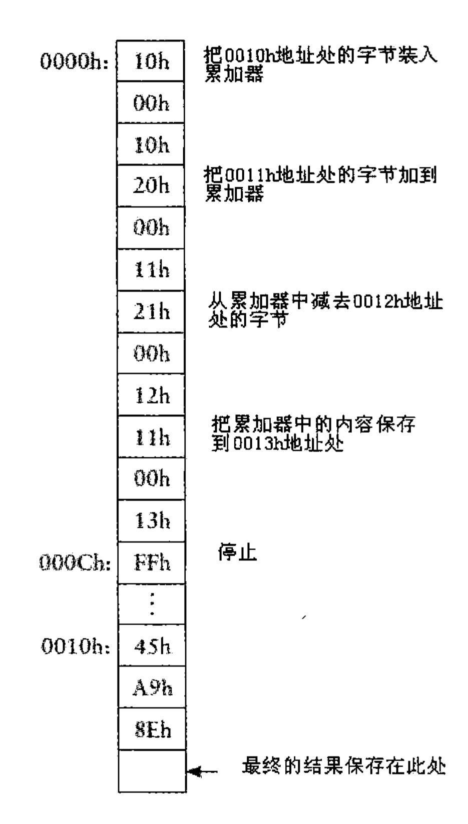
一个 1 KB 的存储器。如果能确定所有的数据都可以存放在 0000h ~ 03FFh 的地址空间中，那么使用一个小于 64 KB 的存储器加法器也可以很好地工作。

但是你现在仍然可能不在意使用了 2 个 RAM 阵列。事实上，确实不必要在意。前面介绍了两种 RAM 阵列，一个用来存放指令码，另一个用来存放操作数据——这种设计使得自动加法器的结构非常清晰和易于使用。但现在我们使用 3 字节长的指令格式，第二个和第三个字节用来指明操作数的存储地址，因此就没有必要再使用两个独立的 RAM 阵列。操作码和操作数可以存放在同一个 RAM 阵列。

为了实现这个设计，我们需要一个 2-1 选择器来确定如何对 RAM 阵列寻址。通常，和前面的方式相同，我们用一个 16 位的计数器来计算地址。数据 RAM 阵列的输出仍然连接到 3 个锁存器，分别用来保存指令代码及其对应操作数的 16 位地址，其 16 位的地址输出是 2-1 选择器的第二种输入。地址被锁存后，可以通过选择器将其作为 RAM 阵列的地址输入。



我们已经对原电路做了不少改进，现在可以把操作指令和操作数据保存在同一个 RAM 阵列中。例如，下图演示了如何把两个 8 位数相加，然后从结果中再减去一个 8 位数。

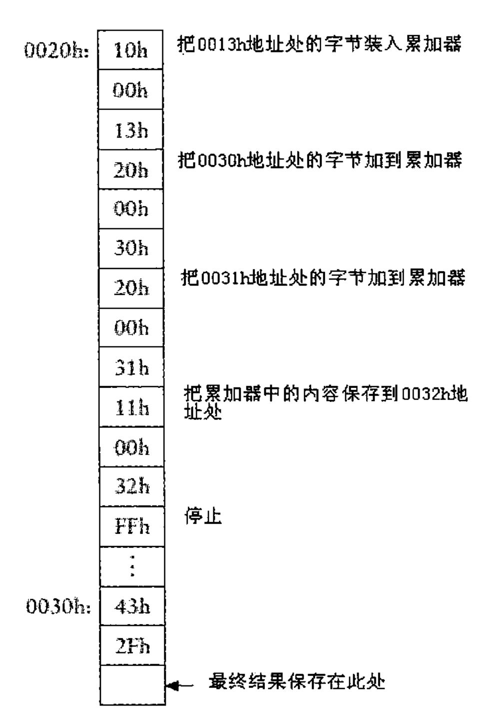


通常，指令从 0000h 开始存放，这是因为当计数器复位后从该位置访问 RAM 阵列。最后的 Halt 指令存放在 000Ch 地址。我们可以把这 3 个操作数及它们的运算结果保存在 RAM 阵列的任何地址（当然这不包括最开始的 13 个字节，因为它们已经用来存放操作指令），所以我们选择在从 0010h 地址开始保存操作数。

假设现在你发现需要在原来的结果中再加两个数，你可以向存储器中输入一些新的指令以替换原来所有的指令，但是你可能不愿意这么做。或许你更倾向于在原指令的地址后增加一些新的指令。第一步要做的就是把 000Ch 地址处的 Halt 指令替换为一个 Load 指令。但你仍然需要增加两条 Add 指令，一条 Store 指令，以及一条新的 Halt 指令。唯一的问题是，现在 0010h 地址已经保存了一些数据，因此需要把这些数据转移到较高的地址空间中，然后还需要修改那些指向这些地址空间的指令。

试想一下，把操作码和操作数存放在同一个 RAM 阵列并不是一个急于解决的问题。但可以肯定的是，这是一个迟早要解决的问题，不如现在就找一个解决办法吧。在当前

的例子中，也许你更愿意从 0020h 地址开始存放新的指令，并从 0030h 处开始存放新的操作数据。



注意，第一条 Load 指令所指向的地址为 0013h，这个位置保存着第一次运算的结果。

现在，两部分指令的位置分别起始于地址 0000h 和 0020h，而两部分操作数据的地址分别起始于 0010h 和 0030h。我们希望自动加法器从 0000h 开始执行所有指令完成计算任务。

我们必须移除 000Ch 处的 Halt 指令，这里的移除是指用其他代码替换它。但仅仅如此是不够的，问题在于不论我们用什么来替换 Halt 指令，保存在地址 000Ch 的字节都会被当做指令代码。更糟糕的是，从这个位置开始，存储器中每隔 3 个字节的地址：000Fh，0012h，0015h，0018h，001Bh 以及 001Eh，这些地址保存的字节也会被当做指令代码来处理。如果其中的一个字节恰好为 11h（这是 Store 指令的代码）将会发生什么？如果 Store 指令后面的 2 个字节的地址所指向的位置刚好为 0023h，那又会发生什么呢？它导致的结果是，加法器将累加器中的内容写入这个地址，而这个地址中已经保存了重要的数据。即使这些情况都没有发生，加法器从存储器的 001Eh 地址之后取到的下一条指令的位置

将是 0021h，不是 0020h，而事实上 0020h 才是下一条指令的存储地址。

要是保留 000Ch 地址处的 Halt 指令，我们能寻求到更好的解决办法吗？

不过，我们可以用一个称为 Jump（跳转）的新指令来替换 Halt 指令。现在把它加入到指令表。

操作码	代码
Load	10h
Store	11h
Add	20h
Subtract	21h
Add with Carry	22h
Subtract with Borrow	23h
Jump（跳转）	30h
Halt	FFh

通常情况下自动加法器是以顺序方式对 RAM 阵列寻址的。Jump 指令改变了机器的这种寻址方式，取而代之的是从某个指定的地址开始寻址。这种指令有时也被称作分支（branch）指令或者 Goto 指令，即"转到另一个位置"。

在上面的例子中，我们可以用一个 Jump 指令来替换 000Ch 地址处的 Halt 指令。

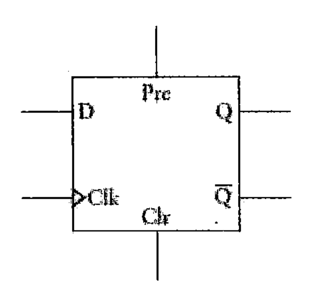
地址	内容	说明
000Ch:	30h	跳转到位于0020h地址处的指令
	00h	
	20h	

30h 即 Jump 指令的代码。其后的两个字节中存放的 16 位地址就是自动加法器要执行的下一条指令的地址。

因此在上面的例子中，自动加法器仍然从 0000h 地址开始，依次执行一条 Load 指令，一条 Add 指令，一条 Subtract 指令和一条 Store 指令。之后执行一条 Jump 指令，跳转至地址 0020h 继续依次执行一条 Load 指令，两条 Add 指令，一条 Store 指令，最后执行一条 Halt 指令。

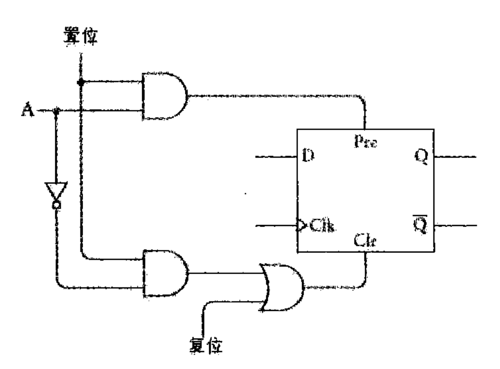
Jump 指令通过作用于 16 位计数器实现其功能。无论何时，只要自动加法器遇到 Jump

指令，计数器就会被强制输出该 Jump 指令后的 16 位地址。这可以通过 16 位计数器的 D 型边沿触发器的预置（Pre）和清零（Clr）输入来实现：



这里要再次声明，在正常的操作下，Pre 和 Clr 端的输入都应该是 0。但是，当 Pre=1，Q=1；当 Clr=1，则 Q=0。

如果你希望向一个触发器加载一个新的值（用 A 表示，代表地址），可以像下图所示这样连接。

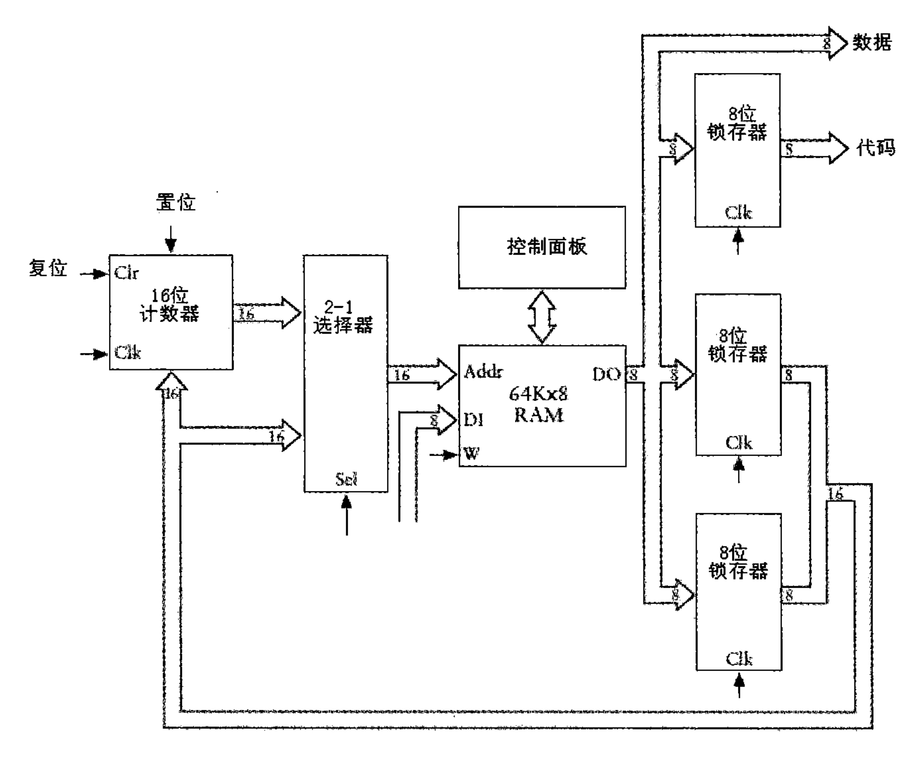


通常，置位信号为 0。此时，触发器的预置端输入为 0，在复位信号不为 1 的情况下，清零信号也为 0。在这种情况下，触发器就可以独立清零，而不受置位信号的影响。当置位信号为 1 时，如果 A 为 1，则清零输入为 0，预置输入为 1；如果 A 为 0，则预置输入 0，清零输入为 1。这就意味着 Q 端将被设置为与 A 端相同的值。

我们需要为 16 位计数器的每一位设置一个这样的触发器。一旦加载了某个特定的值，计数器就会从该值开始计数。

然而，这对电路的改动并不是很大。从 RAM 阵列锁存得到的 16 位地址既可以作为 2-1 选择器（它允许该地址作为 RAM 阵列的地址输入）的输入，也可以作为 16 位计数器

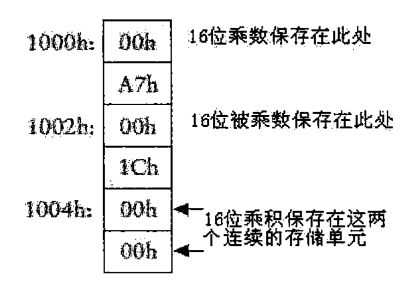
置位信号的输入。



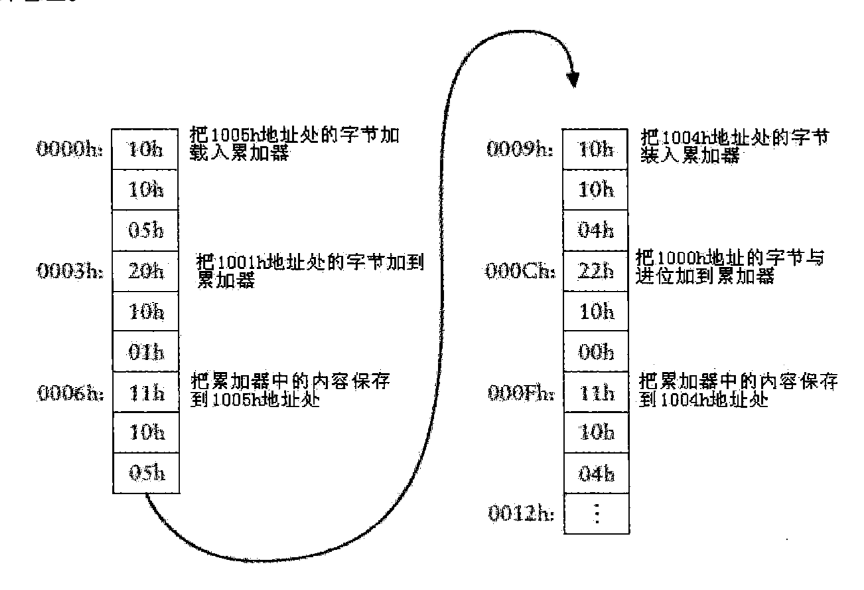
显然，只有当指令代码为 30h 并且其后的 16 位地址被锁存时，我们才必须确保置位信号为 1。

毋庸置疑，Jump 指令的确很有用。但与之相比，一个在我们想要的情况下跳转的指令更加有用，这种指令称做条件跳转（Conditional Jump）。也许说明该命令重要性的最好方法是这样一个问题：怎样让自动加法器进行两个 8 位数的乘法运算？例如，我们如何利用自动加法器得到像 A7h 与 1Ch 相乘这种简单运算的结果呢？

这其实很简单。两个 8 位数相乘得到一个 16 位数，为了方便起见，把该乘法运算中涉及的 3 个数均表示为 16 位数。第一步确定要把乘数和乘积保存到什么位置。



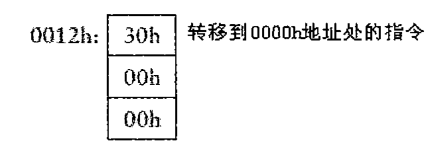
大家都知道 A7h 和 1Ch 相乘的结果（即十进制的 28）和把 28 个 A7h 累加的结果相同。因此保存在地址 1004h 和 1005h 的字节实际上是累加的结果。下图演示了如何把 A7h 加到该地址。



当这 6 条指令执行完毕之后，存储器 1004h 和 1005h 地址保存的 16 位数与 A7h 乘以 1 的结果相同。因此，为了使存放于该地址的值等于 A7h 与 1Ch 相乘的结果，要把这 6 条指令再反复执行 27 次。为了达到这个目的，可以在 0012h 地址开始把这 6 条指令连续输入 27 次；也可以在 0012h 处保存一个 Halt 指令，然后将复位键连续按 28 次得到最终结果。

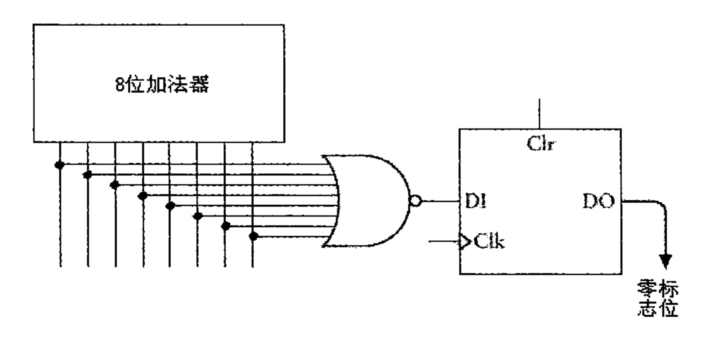
当然，这两种方式都不是很理想。它们都要求你做重复做许多遍烦琐的事情：输入一系列指令或者反复按复位键，而重复的次数就等于乘数。你当然是不会愿意用这种方式来进行 16 位数乘法运算的。

但如果在地址 0012h 处放置一条 Jump 指令会怎样呢？这个指令使得计数器再次从 0000h 处开始计数。



这的确是一个巧妙的方法。第一次执行完指令之后，位于存储器的 1004h 和 1005h 地址的 16 位数等于 A7h 乘 1，然后 Jump 指令使下一条指令从存储器顶部开始执行。第二次执行指令后，该 16 位数等于 A7h 乘 2，最后其结果可以等于 A7h 乘 1Ch。但是，这个过程不会停止下来，它会一直反复执行下去。

我们需要的是这样一种 Jump 指令，它只让这个过程重复执行所需要的次数，这种指令就是条件跳转指令，它并不难实现。要实现它，要做的第一步是增加一个与进位锁存器类似的 1 位锁存器。该锁存器被称为零锁存器（Zero latch），这是因为只有当 8 位加法器的输出全部为 0 时，它锁存的值才是 1。



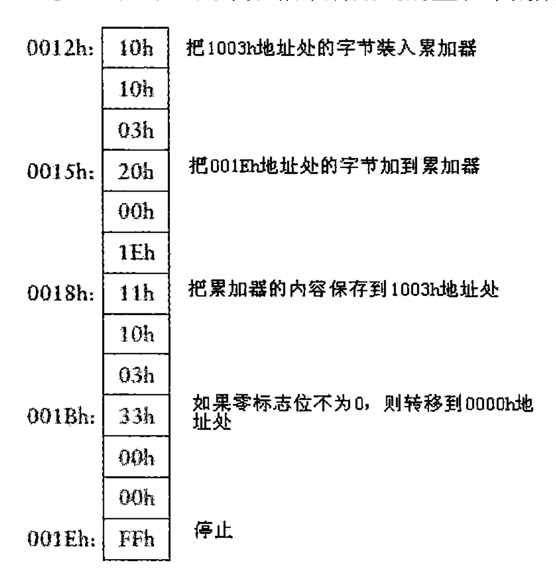
使或非门的输出为 1 的唯一方法是其所有的输入全为 0。与进位锁存器的时钟输入一样，只有当 Add、Subtract、Add with Carry、Subtract with Borrow 这些指令执行时，零锁存器才锁存 1 个数，该数称做零标志位（Zero flag）。注意，它是以一种似乎是相反的方式工作的：当加法器的输出全为 0 时，零标志位等于 1；当加法器的输出不全为 0 时，零标志位等于 0。

有了进位锁存器和零锁存器以后，我们可以为指令表新增 4 条指令。

操作码	代码
Load	10h
Store	11h
Add	20h
Subtract	21h
Add with Carry	22h
Subtract with Borrow	23h
Jump	30h
Jump If Zero（零转移）	31h
Jump If Carry（进位转移）	32h
Jump If Not Zero（非零转移）	33h
Jump If Not Carry（无进位转移）	34h
Halt	FFh

例如，非零转移指令（Jump If Not Zero）只有在零锁存器的输出为 0 时才会跳转到指定的地址。换言之，如果上一步的加法、减法、进位加法、或者借位减法等运算的结果为 0 时，将不会发生跳转。为了实现这个设计，只需要在常规跳转命令的控制信号之上再加一个控制信号：如果指令是 Jump If Not Zero，那么只有当零标志位是 0 时，16 位计数器才被触发。

下图中 0012h 地址之后的指令即两个数相乘所用到的上表中的所有指令。



第一次循环之后，位于地址 0004h 和 0005h 处的 16 位数等于 A7h 与 1 的乘积，这和我们的设计相符。在上图中，地址 1003h 处的字节通过 Load 指令载入到累加器，该字节是 1Ch。把这个数和 001Eh 地址的字节相加后，恰好遇到的是 Halt 指令，当然这是一个合法的数。FFh 与 1Ch 相加的结果与从 1Ch 中减去 1 的结果相同，都是 1Bh，因为这个数不等于 0，所以零标志位是 0，1Bh 这个结果会存回到 1003h 地址。下一条要执行的指令是 Jump If Not Zero，零标志位没有置为 1，因此发生跳转。接下来要执行的一条指令位于 0000h 地址。

需要记住的是，Store 指令不会影响零标志位的值。只有 Add、Subtract、Add with Carry、Subtract with Borrow 这些指令才能影响零标志位的值，因此它的值与最近执行上述某个指令时所设置的值相同。

经过两次循环后，1004h 和 1005h 地址所保存的 16 位数等于 A7h 与 2 的乘积。1Bh 与 FFh 的和等于 1Ah，不为 0，因此仍然返回到顶部执行。

当执行到第 28 次循环时，1004h 和 1005h 地址保存的 16 位数等于 A7h 和 1Ch 的乘积。1003h 地址保存的值是 1，它和 FFh 相加的结果是 0，因此零标志位被置位！Jump If Not Zero 指令不会再跳转到 0000h 地址，相反，下一条要执行的指令即 Halt 指令。这样，我们就完成了全部的工作。

现在可以断言，我们一直不断完善的这组硬件构成的机器确实可以被称为计算机（computer）。当然，它还很原始，但毕竟是一台真正的计算机。条件跳转指令将它与我们以往设计的加法器区别开来，能否控制重复操作或者循环（looping）是计算机（computer）和计算器（calculator）的区别。这里演示了该机器如何用条件跳转实现两个数的乘法运算，用类似的方法还可以进行两个数的除法运算。而且，这不仅仅局限于 8 位数，它可以对 16 位、24 位、32 位，甚至更高位的数进行加、减、乘、除运算。而且，既然它能完成这些运算，那么对于开平方根、取对数、三角函数等运算也完全可以胜任。

既然我们已经装配了一台计算机，因此可以使用计算机相关词汇了。

我们装配的计算机属于数字计算机（digital computer），因为它只处理离散数据。曾经还有一种模拟信号计算机（analog computer），但现在已经非常少见了。（数字数据就是离散数据，即这些数据是一些确定的离散值；模拟数据是连续的，并且在整个取值区间

变化。)

一台数字计算机主要由 4 部分构成：处理器（processor）、存储器（memory），至少一个输入（input）设备和一个输出（output）设备。我们装配的计算机中，存储器是 64 KB 的 RAM 阵列，输入和输出设备分别是 RAM 阵列控制面板上的开关和灯泡。这些开关和灯泡可以让我们向存储器中输入数据，并可以检查运算结果。

除了上述 3 种设备之外的其他设备都归类于处理器。处理器也被称作中央处理单元（central processing unit）或者 CPU。更通俗的说法是将其称作计算机的大脑，但本文将避免使用这样的词，因为我们所设计的处理器称不上是大脑（今天，微处理器这个词使用得非常普遍，它是一种非常小的处理器，可以通过本书第 18 章讲到的技术来构造它。但本章中通过继电器构造的机器无论如何也称不上"微小"的）。

我们设计的处理器为 8 位处理器。累加器的宽度为 8 位，而且大部分数据通路都是 8 位的宽度。唯一的 16 位数据通路是 RAM 阵列的地址通路。如果该通路也采用 8 位宽度的话，存储器容量最多就只有 256 字节，而不再是 65536 字节，这样处理器的功能会受到很大的限制。

处理器包括若干组件。毫无疑问累加器就是其中一个，它只是一个简单锁存器，用来保存处理器内部的部分数据。在我们所设计的计算机中，8 位反相器和 8 位加法器一起构成了算术逻辑单元（Arithmetic Logic Unit），即 ALU。该 ALU 只能进行算术运算，最主要的是加法和减法运算。在更加复杂的计算机中（我们会在后面的章节看到），ALU 还可以进行逻辑运算，比如"与"（AND），"或"（OR），"异或"（XOR）等。16 位的计数器被称做程序计数器（PC，Program Counter）。

我们的计算机是由继电器、电线、开关，以及灯泡构造而成的，这些东西都叫做硬件（hardware）。与之对应，输入到存储器中的指令和数值被称做软件（software）。之所以把"硬"改成了"软"，是因为相对于硬件而言，指令和数据更容易修改。

当我们在计算机领域进行讨论时，"软件"这个词几乎与"计算机程序"（computer program），或"程序"（program）等术语是同义的，编写软件也称为计算机程序设计（computer programming）。我们确定用一些指令让计算机实现两个数相乘的过程就是在进行计算机程序设计。

通常，在计算机程序中，我们能够把代码（即指令本身）和数据（即代码要处理的数）区别开。但有时它们之间的界限也不是很明显，比如 Halt 指令（FFh）就可以有两种功能，除了作为代码时表示停止执行外还能代表数值–1。

计算机程序设计有时也被称做编写代码（writing code），或编码（coding），也许你经常会听到："我整个假期都在编码"，"我一直干到今天早上，敲出了很多行代码"。计算机程序设计人员有时也被称做编码员（coders），尽管有些人可能认为这是一个贬义词。程序员更喜欢被别人称做"软件工程师"（software engineers）。

能够被处理器响应的操作码（比如 Load 指令和 Store 指令的代码 10h 和 11h），称做机器码（machine codes），或机器语言（machine language）。计算机能够理解和响应机器码，其原理和人类能够读写语言是类似的，因此这里使用了"语言"来描述它。

一直以来，我们都在使用很长的短语来引用机器所执行的指令，比如 Add with Carry 指令。通常而言，机器码都分配了对应的简短助记符，这些助记符都用大写字母表示，包括 2 个或 3 个字符。下面是一系列上述计算机大致能够识别的机器码的助记符。

操作码	代码	助记符
Load（加载）	10h	LOD
Store（保存）	11h	STO
Add（加法）	20h	ADD
Subtract（减法）	21h	SUB
Add with Carry（进位加法）	22h	ADC
Subtract with Borrow（借位减法）	23h	SBB
Jump（转移）	30h	JMP
Jump If Zero（零转移）	31h	JZ
Jump If Carry（进位转移）	32h	JC
Jump If Not Zero（非零转移）	33h	NC
Jump If Not Carry（无进位转移）	34h	JNC
Halt	FFh	HLT

当这些助记符与另外一对短语结合使用时，其作用更加突出。例如，对于这样一条长语句"把 1003h 地址处的字节加载到累加器"，我们可以用如下简洁的句子替代：

```
LOD A, [1003h]
```

位于助记符右侧的 A 和[1003h]称为参数（argument），它们是这个 Load 指令的操作对象。参数由两部分组成，左边的操作数称为目标（destination）操作数（A 代表累加器），右边的操作数称为源（source）操作数。方括号“[]”表明要加载到累加器的不是 1003h 这个数值，而是位于存储器地址 1003h 的数值。

类似的，指令“把 001Eh 地址的字节加到累加器”，可以简写为：

```
ADD A, [0010Eh]
```

指令“把累加器中的内容保存到 1003h 地址”，可简写为：

```
STO [1003h], A
```

注意，在上面的语句中，目标操作数（Store 指令在存储器中的位置）仍然在左边，源操作数在右边。这就决定了累加器中的内容必须要保存到存储器的 1003h 地址。“如果零标志位不是 1 则跳转到 0000h 地址处”这个冗长的语句可以简明地表示为：

```
JNZ  0000h
```

注意，这里没有使用方括号，这是因为跳转指令要转移到的地址是 0000h，而不是保存于 0000h 地址的值，即 0000h 地址就是跳转指令的操作数。

用缩写的形式表示指令是很方便的，因为在这种形式下指令以可读的方式顺序列出而不必画出存储器的空间分配情况。通过在一个十六进制地址后面加一个冒号，可以表示某个指令保存在某个特定地址空间，例如：

```
0000h:        LOD A, [1005h]
```

下面的语句表示了数据在特定地址空间的存储情况。

```
1000h:        00h, A7h
1002h:        00h, 1Ch
1004h:        00h, 00h
```

你可能已经注意到了，上面的两个字节都是以逗号分开的，它表示第一个字节保存在左侧的地址空间中，第二个字节保存在该地址后的下一个地址空间中。上面的三条语句等价于下面的这条语句：

```
1000h:        00h, A7h, 00h, 1Ch, 00h, 00h
```

因此上面讨论的乘法程序可以用如下一系列语句来表示：

```
0000h:      LOD A, [1005h]
            ADD A, [1001h]
            STO [1005h], A

            LOD A, [1004h]
            ADC A, [1000h]
            STO [1004h], A

            LOD A, [1003h]
            ADD A, [001Eh]
            STO [1003h], A

            JNC 0000h

001Eh:      HLT

1000h:      00h, A7h
1002h:      00h, 1Ch
1004h:      00h, 00h
```

使用空格和空行的目的仅仅是为了人们更方便地阅读程序。

在编码时最好不要使用实际的数字地址，因为它们是可变的。例如，如果要把数值保存在存储器的 2000h ~ 2005h 地址空间中，你需要在程序中重复多次写这些语句。用标号（label）来指代存储器中的地址空间是个较好的办法。这些标号是一些简单的单词，或是类似单词的字符串。上面的代码可以改写为：

```
BEGIN:      LOD A, [RESULT + 1]
            ADD A, [NUM1 + 1]
            STO [RESULT + 1], A

            LOD A, [RESULT]
            ADC A, [NUM1]
            STO [RESULT], A

            LOD A, [NUM2 + 1]
            ADD A, [NEG1]
            STO [NUM2 + 1], A

            JNZ BEGIN
```

```
NEG1:        HLT

NUM1:        00h, A7h
NUM2:        00h, 1Ch
RESULT:      00h, 00h
```

注意，NUM1，NUM2，RESULT 这些标号都是指存储器中保存两个字节的地址单元。在这些语句中，NUM1+1，NUM2+1 和 RESULT+1 分别指标号 NUM1，NUM2，RESULT 后的第二个字节。注意，NEG1（negative one）用来标记 HLT 指令。

最后，如果你可能忘记这些语句所表示的意思，那么可以在该语句后面加注释（comment），这些注释可以用我们人类的自然语言表述，然后通过分号与程序语句分隔开。

```
BEGIN:       LOD A, [RESULT + 1]
             ADD A, [NUM1 + 1]       ; 低字节相加
             STO [RESULT + 1], A

             LOD A, [RESULT]
             ADC A, [NUM1]           ; 高字节相加
             STO [RESULT], A

             LOD A, [NUM2 + 1]
             ADD A, [NEG1]        ; 第二个数减 1
             STO [NUM2 + 1], A

             JNZ BEGIN

             NEG1:        HLT

             NUM1:        00h, A7h
             NUM2:        00h, 1Ch
             RESULT:      00h, 00h
```

这里给出的是一种计算机程序设计语言，称为汇编语言（assembly language）。它是全数字的机器语言和指令的文字描述的一种结合体。同时它用标号表示存储器地址。人们有时候会混淆机器语言和汇编语言，这是因为它们是对同一种事物的不同描述方式。每一条汇编语句都对应着机器语言中的某些特定字节。

如果你想为本章所设计的计算机编写程序，那么可能首先想到的是用汇编语言来编

写（在纸上）。在你确定程序无误并准备验证其运行结果的时候，你需要手工对其汇编：这就意味着要把每一条汇编语句转换成与之对应的机器语言，这仍然要在纸上操作。完成之后，你需要通过开关把这些机器码输入到 RAM 阵列中并运行该程序，也就是让计算机执行这些指令。

对于学习计算机程序设计的人来说，应该尽早了解“错误”（bug）这个术语。当你编写代码时——特别是采用机器语言——是非常容易出错的。输入一个错误的操作数已经很不妙了，如果输错的是一个指令代码的话，情况会怎样呢？当你准备输入 10h（Load 指令）的时候，却输入了 11h（Store 指令），造成的后果是：期望的数据不会被机器加载，而该地址的数据还会被累加器中的内容替换掉。

一些错误可能导致意想不到的结果。假设你使用 Jump 指令跳转到一个地址，而该地址没有存放任何合法的指令，或者你偶然误用 Store 指令覆盖了其他指令，类似的情况都有可能发生（而且会经常发生）。

甚至上述乘法程序中就存在着一个错误。如果你把程序执行两次，第二次得到的将会是 A7h 与 256 相乘的结果，并且程序会把这个结果与第一次运算的结果相加。这是因为程序执行一次之后，1003h 地址保存的数值是 0。当第二次执行时，FFh 与这个 0 相加的结果不是 0，因此程序会继续执行直到它变为 0。

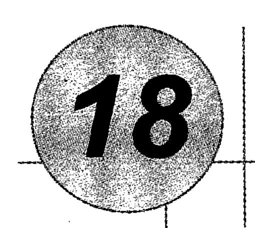
我们已经利用该机器完成了乘法运算，用类似的方法也可以进行除法运算。同时，前面也讲过，利用这些基本功能还可以进行平方根、对数、三角函数等运算。机器所需要的仅仅是能够做加、减法的硬件以及利用条件跳转指令执行代码的方法。正如程序员经常挂在嘴边的一句话：“我可以用软件完成其他工作”，这些工作我们都可以编程实现。

当然，软件可能是很复杂的。有很多专门讲授程序员如何用算法（algorithm）解决特殊问题的书，本书不打算讲这些内容。目前我们一直讨论的都是自然数，并没有考虑如何在计算机中表示十进制小数，本书将在第 23 章讨论这个问题。

前面不止一次强调过，这些硬件部件早在 100 年前就发明出来了。但是，本章所设计的计算机在当时却并没有被创造出来。当继电器计算机在 20 世纪 30 年代中期被设计出来的时候，很多包含在其中的概念还并不为人所知，直到 1945 年左右世人才开始慢慢了解它们。例如，在当时，人们仍然尝试在计算机中使用十进制数而不是二进制数；而

且计算机程序也不是完全存储在存储器中的，有时候它们会被保存在纸带上。特别是在早期的计算机中，存储器非常昂贵并且体积庞大，不论在 100 年前还是在今天，用 500 万个电报继电器构建一个 64 KB 的 RAM 阵列都是令人感到荒唐的事。

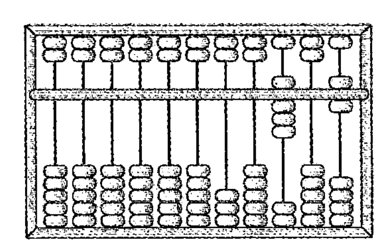
当回顾完计算器和计算机这一段历史，让我们展望一下未来。或许有一天我们会发现：没有必要建造一个如此复杂的继电器计算机。正像在第 12 章所讨论过的那样，继电器最终会被真空管和晶体管这类电子器件所替代。或许我们还会发现，已经有人创造出一种全新的设备，它的处理器及存储器的能力与我们所设计出的不相上下，但是，这种机器精致小巧，甚至可以放在我们的掌心。



从算盘到芯片

自古以来，人们为了尽量简化数学计算，绞尽脑汁发明了很多精巧的工具和机器。虽然人类的计数能力与生俱来，但需要帮助是在所难免的。每个人都是各有所长、各有所短，所以经常会遇到一些自身无法解决的问题。

在人类社会早期，人们借助数字这种工具来帮助自己记录物品和财产。包括古希腊以及美洲土著在内的很多文化中，人们借助小卵石或者谷粒进行计数。在欧洲，这些古老的计数方式演变成了计数板，而在中东，则演变成为我们较为熟悉的由骨架和算珠组成的算盘（abacus），算盘的样子就像下面这幅图所示。



尽管人们通常将算盘与亚洲文化，尤其是中国文化联系到一起，但它似乎是在公元 1200 年左右由商人带到中国的。

很多人不喜欢做乘法和除法，但仍然有一小部分人对此进行了一些研究。苏格兰数

学家约翰·纳皮尔（John Napier，1550-1617）就是这小部分人中的一员，为了简化某些操作，他发明了对数。例如两个数乘积就可以简单地表示为其对数之和。因此，如果要得到两个数的乘积，可以采取以下步骤：首先在对数表中查出这两个数的值，然后将其值相加，最后在对数表中逆向搜寻出其乘积。

在随后的 400 年时间里，一些最伟大的思想家一直致力于建立对数表这项工作，与此同时，另外一些人则设计出一些小装置期望代替对数表。其中，一种带对数刻度的滑尺久负盛名，它由埃德蒙·甘特（Edmund Gunter，1581-1626）发明，而威廉·奥特雷德（William Oughtred，1574- 1660）对其进行了改进。1976 年当克鲁夫&艾萨（Keuffel & Esser）公司将其制造的最后一个滑尺赠送给华盛顿史密斯索尼亚（Smithsonian）学院时，滑尺也就宣布退出了历史舞台，导致其退出的原因是手持计算器的出现。

纳皮尔发明了另外一种由刻在骨头、牛角、象牙上的数字条组成的乘法辅助器，称为纳皮尔骨架（Napier's Bones）。在 1620 年左右，威廉·斯奇卡（Wilhelm Schickard，1592-1635）制造出了最早的机械计算器，它类似于已经初步具备了自动功能的纳皮尔骨架。由相互连接的车轮、齿轮以及水平仪组成的一种计算器几乎在同一时期出现，身为数学家以及哲学家的布莱兹·帕斯卡（Blaise Pascal，1623-1662）和哥特福瑞德·武赫勒姆·范·莱布尼兹（Gottfried Wilhelm von Leibniz，1646-1716）是这种机械计算器的两个最主要的发明者。

你一定对 8 位加法器以及能对多于 8 位的数进行运算的加法器中的进位过程记忆犹新，因为这实在是太烦琐了。起初，进位仅被看做是加法运算中的“一碟小菜”，但它却成了加法器的核心问题。换句话讲，尽管我们设计了一个除了进位功能以外其他功能都俱全的加法器，这样的器件离“大功告成”还是很远！

评价老式计算器的一个关键是其进位处理能否成功，例如帕斯卡的设计，他的进位机制禁止进行减法运算，在进行减法运算时，采用的是加上 9 的补数的方式（在第 13 章中介绍过）。直到 19 世纪后半叶，真正意义上的机械计算器才得以出现并为人们所使用。

约瑟夫·玛丽·杰奎德（Joseph Marie Jacquard，1752-1834）发明的一种奇妙的自动织布机对计算的历史产生了深远的影响，其程度甚至不亚于其在纺织行业产生的影响。杰奎德织布机（大约于 1801 年出现）使用打孔的金属卡片（很像钢琴上面的金属卡片）控制织物上的图案。杰奎德使用了大约 1 万张卡片完成了一幅杰作，那就是用黑白线织

成的自画像。

在 18 世纪（直到 20 世纪 40 年代），计算机就好比一个以计算维持生计的人，而计算能力就好比计算机的生命线。在这个用天上星辰进行航海导航的时期，经常要用到对数表，而三角函数表对航海导航也非常重要。此外，如果你想要发表新数学表，就需要使用许多台计算机，让它们一起工作，最后将所有的结果汇总成一张表。当然，从初始计算到设置打印最终结果，每一个阶段都可能会出现意想不到的错误。



为了消除数学表中错误，英国数学家和经济学家查尔斯·巴贝芝（Charles Babbage，1791-1871，见右图）勤奋工作，他和塞缪尔·莫尔斯（Samuel Morse）差不多是同时代的人。

在那个时期，数学表（例如对数表）并没有计算表中每一项的确切对数值，因为那样做将耗费太多时间。取而代之的方法是选择性的对数计算，即选取一些数字进行对数计算，而对于介于这些数字之间的数的对数则采用插补法进行填充，即差分法（differences），通过相对简单的计算求得结果。

大约在 1820 年，巴贝芝认为他可以设计制造一台可以自动建表，甚至可以自动设置打印类型的机器，这种机器可以完全消除上述错误。因此他设想出了差分机（Difference Engine），从本质上讲差分机是一个大型机械加法器。在差分机中，多位的十进制数通过可以啮合在 10 个不同位置的轮子表示，而负数用 10 的补数来表示。尽管早期一些模型表明巴贝芝的设计是完全可行的，而且他也获得了英国政府的一些资金支持（当然不是很多），但差分机却从来没有完成过，在 1833 年，巴贝芝放弃了这项工作。

可正在那个时候，巴贝芝有了一个更好的想法，那就是解析机（Analytical Engine），他的后半生一直都在不断重复地设计与修改（其间还制作过几个小模型以及部分构件）这个机器，直到其生命的尽头。解析机是 19 世纪最接近于计算机的器件，在巴贝芝的设计中，解析机包含一个存储部件（类似于现在存储器的概念）和一个运算部件（类似于算术逻辑单元）。乘法可以通过重复的加法运算求解，同样的，除法可以通过重复的减法求解。

解析机最令人着迷的地方在于，可以使用改造的杰奎德织布机中的卡片来编程。正

如奥古斯塔·艾达·拜伦（Augusta Ada Byron，1815-1852），即拉弗雷斯女伯爵对解析机的评价（这句话出于她翻译的一篇由意大利数学家撰写的关于巴贝芝解析机的文章）："我们可以肯定地说，正如杰奎德提花织布机织出了花瓣和树叶，解析机编织出了代数的结构模型。"

巴贝芝大概是第一个意识到条件跳转在计算机中重要性的人，关于这个问题，奥古斯塔·艾达曾写下这样一段话"操作循环（cycle），应该这样去理解：

> 它意味着某个操作集（set of operations）重复执行的次数不止一次。它的次数可以是仅仅两次或者是无限次，但它们实际上都是组成操作集的操作被重复执行了。在很多实例的分析中，我们经常会看到由一个或多个循环构成的重复组群（recurring group），也就是循环中包含的一个循环或者多个循环。"

尽管在 1853 年差分机最终由一对父子——乔治（George）和爱德华·舒尔茨（Edvard Scheutz）制造出来，但已经被人们遗忘了好多年的巴贝芝设计的差分机，直到 20 世纪 30 年代才因为人们开始探索 20 世纪计算机的起源时而重新被提起。那时，巴贝芝所做的一些工作已经都被后来的技术超越，除了超前的自动化观念，他所做的工作对 20 世纪计算机工程来说几乎没有可以利用的。

计算史上另一个转折点源于美利坚合众国宪法第一条第二款。这一款其中要求每 10 年进行一次人口普查。1880 年的人口普查要求登记居民的年龄、性别以及国籍信息，每一次人口普查的数据采集工作都要花费大约 7 年的时间。

人口普查局的官员们担心 1890 年的人口普查的数据采集和处理可能会花费 10 年以上的时间，因此他们研究了在工作中使用自动系统的可能性，并选择了由赫尔曼·霍尔瑞斯（Herman Hollerith，1860-1929）开发的机器，他曾在 1880 年人口普查中进行过相关工作。



霍尔瑞斯的设计需要使用大小为 $6\frac{5}{8}\times3\frac{1}{4}$ 寸马尼拉（manila）穿孔卡片（尽管霍尔瑞斯不了解巴贝芝是怎样在解析机中利用卡片编制程序的，但他对杰奎德织布机中如何使用卡片却非常熟悉）。卡片上的孔按 24 列每列 12 个位置排列，共计 288 个位置。这些点代表了在人口普查中需要记录的一

个人的特征，人口普查工作人员通过在卡片适当的位置上打 1/4 英寸的方孔来标记上述的特征。

在读本书时你可能会习惯性地想到二进制编码，而且你或许立刻会猜想，可以利用卡片上的 288 个穿孔来存储 288 位的信息，但是卡片的使用方式并非如此。

举例来说，在纯二进制系统中使用的人口普查卡片上应该有一个位置代表性别，在该位置穿孔则代表男性，不穿孔则代表女性（或与此相反）。但是在霍尔瑞斯设计的卡片中使用了两个位置代表性别，其中第一位置穿孔代表男性，另外一个位置穿孔代表女性。同样的，人口普查工作者通过两个穿孔来标识普查对象的年龄，第一个穿孔指定一个以 5 年为间隔的范围：如 0～4，5～9，10～14，等等。第二个穿孔则表示对象处于该年龄段（5 年）中的哪一年，即可推算出对象的准确年龄。对年龄进行编码共计需要 28 个打孔位置，而在纯二进制系统中只需要 7 个位置就可以对 0～127 的所有年龄进行编码。

霍尔瑞斯在记录人口普查信息时没有中使用二进制系统是可以理解的：对于 1890 年人口普查工作者来说，将年龄转换为二进制数字，这个要求太高了。穿孔卡片系统不能是完全的二进制系统，这里还有一个实际的原因，真正的二进制系统可以产生（几乎）所有孔都被打穿的情况，这将使得卡片极易碎裂，而且看上去也不美观。

可以对人口普查的数据做统计分析或将其制作成表格（tabulated）。比如，你希望了解每一个人口普查行政区中居住着多少人，当然，人口的年龄段的分布也是一个比较令人感兴趣的信息。为了达到这些目的，霍尔瑞斯制造了制表机，它是组合了人工操作以及自动功能的半自动化工具。操作人员把一个有 288 个弹簧针的板子压到每一个卡片上，每一根弹簧针对应于卡片上的一个穿孔位置，当弹簧针与水银池中的水银接触时，形成通路，这个电路触发电磁体即可进行十进制计数。

霍尔瑞斯在卡片分类机上也使用了电磁体。例如，如果需要收集所统记的每一种职业中人员的年龄分布信息，首先需要将卡片按照职业进行分类，然后分别对每种职业中人员的年龄信息进行统计。分类机使用和制表机中一样的手压方式，不同之处在于分类机使用的电磁体可以将 26 个间隔区域中任意一个的舱口打开。操作者把对应的卡片通过舱口放入间隔区域，之后再手动关闭舱口。

在 1890 年人口普查中使用这种自动化技术的实验取得了令人瞩目的成就，在这个实

验中总共加工处理了超过 6200 万张卡片，数据量是 1880 年人口普查的两倍，但时间却只用了后者的三分之一，自此，霍尔瑞斯和他的发明闻名四海。1895 年，他还到访了莫斯科，俄国人欣喜地购买了他的设备，并于 1897 年将其首次应用在俄国人口普查中。

赫尔曼·霍尔瑞斯此后一发不可收拾。1896 年，他创办制表机公司（Tabulating Machine Company），租借并出售其穿孔卡片设备。到 1911 年，由于公司的合并，制表机公司更名为计算制表记录公司（Computing-Tabulating-Recording Company），或者叫做 C-T-R 公司。再到 1915 年，托马斯 J·华盛顿（Thomas J. Watson，1874–1956）成为 C-T-R 公司的总裁，他在 1924 年将公司的名字更改为国际商业机器公司（International Business Machines Corporation），即 IBM。

到 1928 年，在 1890 年人口普查中最初使用的卡片逐渐演变为著名的 IBM 卡片，"do not spindle, fold, or mutilate"，这种卡片有 80 列 12 行，使用了将近 50 年，甚至在其后期，还有人把它们叫做霍尔瑞斯卡片。关于这些卡片的遗留问题将在第 20、21 和 24 章中进一步的讲述。

进入到 20 世纪之前，让我们重新审视一下 19 世纪这一百年。因为主题所限，本书更多的关注的是数字性质的发明，其中包括电报、盲人用点字法、巴贝芝机器，以及霍尔瑞斯卡片。而在与数字概念以及相关设备打交道时，你会发现整个世界皆为数字。但是，19 世纪的发现和发明确切的来讲不是数字的。实际上，通过感官所认识的大自然中只有很少一部分是数字的，更多的时候表现为不可分割的整体。

尽管霍尔瑞斯在他的制表机以及分类机中使用了继电器（relays），但是人们直到 20 世纪 30 年代中期才开始用继电器来构建计算机——它们最终被叫做机电化（electromechanical）计算机。在这些机器中使用的继电器不同于一般的电报继电器，后者的主要作用是为了完善电话系统的路由控制。

早期的继电式计算机与上一章中的继电式计算机不是同一个概念（我们随后会学到，从 20 世纪 70 年代开始，这种继电式计算机依靠微处理器进行计算）。需要特别说明的一点，尽管现代计算机内部使用二进制数，但早期的继电式计算机并非如此。

我们的继电式计算机与早期的继电式计算机存在另外一个不同点，那就是在 20 世纪 30 年代，没有人能够疯狂到用继电器制造出 524,288 位的存储器！资金的花费、空间的

占用和能源的耗费使得制造如此大的存储器变得不大可能。可得到的极少存储器也只用来存储中间结果，而程序本身则存储在一些物理媒介上面，例如带穿孔的纸带。实际上，将代码和程序放入到存储器进行处理是后来发明的做法。

下面按时间顺序进行介绍，第一台继电式计算机由康拉德·楚泽（Conrad Zuse，1910–1995）制造，1935 年还是工科学生的他在其父母位于柏林的家中制造了这台机器。这台机器中使用了二进制数，但其早期的版本中使用的是机械存储器而非继电器。楚泽使用老式 35 毫米电影胶片进行穿孔，然后在上面编制程序。

1937 年，贝尔电话实验室（Bell Telephone Laboratories）的乔治·史提必兹（George Stibitz，1904–1995）将一对电话继电器带回了家中，并在他厨房的桌子上连接了一个 1 位加法器，后来他妻子将其称之为 K 机器（K 是厨房“kitchen”的头一个字母），这个实验促使 1939 年贝尔实验室中复数计算机的诞生。

同一时期，哈佛大学研究生霍华德·艾肯（Howard Aiken，1900–1973）要寻找做大量的重复计算的方法，而正是他的这一需求促使哈佛大学与 IBM 合作，并最终在 1943 年创造出一台自动连续可控计算机（Automated Sequence Controlled Calculator ，ASCC），也就是闻名于世的 Harvard Mark I。这是第一台可以打印表格的数字计算机，它最终将查尔斯·巴贝芝的梦想付诸于现实。Mark II 是最大的继电式计算机，使用了 13,000 个继电器。哈佛大学计算机实验室当时的主任是艾肯，也正是他讲授了计算机科学的第一次课。

对于构造计算机来说，继电器不是最完美的设备，因为它们是机械性的，利用金属片的弯曲和伸直状态进行工作，而频繁的工作可能导致其断裂，另外如果接触点之间有污垢或者卡住纸屑，也会导致继电器失效。1947 年发生了一件著名的事故，人们从 Mark II 计算机的一个继电器中发现了一只飞蛾。格蕾丝·莫瑞·赫柏（Grace Murray Hopper，1906–1992）于 1944 加入了艾肯的团队，日后成为了计算机编程语言领域非常著名的人物。他将上面提到的那只飞蛾用带子绑在计算机日志（logbook）上，并在其边上注明“第一个被发现的有生命的 bug”。

真空管（vacuum tube）是一种可以替代继电器的元件，它是由约翰·安布罗斯·弗莱明（John Ambrose Fleming，1849–1945）和李·德·福雷斯特（Lee de Forest，1873–1961）在进行无线电通信连接研究时开发出来的。到 20 世纪 40 年代，真空管已经被广泛应用于放大电话信号，实际上，那时几乎每一个家庭都拥有一台带有发光二极管可控收音机，

它们能放大无线信号，并且把它们变成还原为人们能听见的声音。真空管同样可以通过连接成与门、或门、与非门，以及与或门——这一点很像继电器。

究竟是由继电器还是由真空管组成这些逻辑门并不重要，重要的是这些逻辑门可以被装配组合成加法器、选择器、解码器、触发器，以及计数器。不论真空管何时取代继电器，前面章节中讲述的关于基于继电器部件的一切同样是有效的。

真空管同样存在自身的问题，比如，价格昂贵、耗电量大，以及产生的热量太多。可是，其最大的问题是真空管最终会被烧坏，如同人活一世一样，是无法改变的事实。那时，拥有真空管收音机的人们习惯于定期更换真空管，而电话系统设计时有很多冗余的真空管，所以一个真空管的报废有时并不是什么大事（不管怎样，人们不会期待电话系统是完美无瑕的）。可是在计算机中，当一个真空管烧坏时并不可能立刻被检测到，此外，一台计算机拥有数量巨大的真空管，按统计学来分析，每隔几分钟就会烧坏一个。

用真空管取代继电器的最大好处在于真空管的状态可以在百万分之一秒（μs）内发生转变。真空管状态转变（开关的打开与关闭）的速度比继电器要快 1000 倍，继电器在其最好状态下状态的转变也需要 1ms，即千分之一秒。十分有趣的是在计算机的早期发展中，计算速度并不是主要考虑的问题，而这个时期的计算速度与从纸张或者电影胶片中读取程序的速度有关。由于当时的计算机都按照这种方式构建，因此采用真空管比继电器到底计算速度提升了多少，并不重要。

但是在 20 世纪 40 年代初期，新设计的计算机中真空管开始取代继电器。到 1945 年，真空管已经完全取代了继电器。虽然继电器计算机被称为电动机械计算机，但真空管是第一台电子计算机的基础。

在英国，巨像（Colossus）计算机（1943 年首次投入使用）用来破译德国名为“Enigma”代码生成器产生的代码，艾伦·M·图灵（Alan M. Turing，1912-1954）为这个项目（以及英国后来几个的计算机项目）做出了巨大贡献，图灵撰写了两篇非常有影响的论文，这使他如今成为计算机领域的鼎鼎大名的人物。第一篇论文发表于 1937 年，首次提出了“可计算性”（computability）这个概念，用来分析哪些事情计算机可以做到，哪些做不到。他为计算机构想了一个抽象模型，这就是现在为人所熟知的图灵机（Turing Machine）。图灵第二篇非常有名的论文是关于人工智能的，在这篇论文中他介绍了一种测试机器智能的方法，即现在为人熟知的图灵测试法（Turing Test）。

在摩尔电子工程学院（宾夕法尼亚大学），J·普利斯普·埃克特（J. Presper Eckert，1919-1995）和约翰·莫克利 （John Mauchly，1907-1980）设计了 ENIAC（Electronic Numerical Integrator and Computer，电子数字积分计算机），使用了 18,000 个真空管并最终在 1945 年底完成。按全部吨位算（大约 30 吨），ENIAC 是曾经（或许以后也是）制造出来的最大的计算机。到 1977 年，人们可以在 Radio Shack 买到速度更快的计算机。埃克特和莫克利想为计算机申请专利，可是却被竞争者约翰·V·安塔纳索夫（John V. Atanasoff，1903-1995）阻扰了，他更早一步设计了一台电子计算机，但它运行得并不顺畅。

ENIAC 吸引了数学家约翰·冯·诺依曼（John von Neumann，1903- 1957）的眼球。从 1930 年开始，出生在匈牙利的冯·诺依曼就定居美国。作为一名令公众瞩目的人物，因其仅凭自己的大脑就能进行复杂的数学计算而闻名，冯·诺依曼当时是普林斯顿高级研究院的一名数学教授，研究范围很广，从量子力学到游戏应用，甚至到经济理论。



约翰·冯·诺依曼协助设计的 ENIAC 的后续产品 EDVAC（Electronic Discrete Variable Automatic Computer）。特别是在 1946 年与亚瑟·W·伯克斯 （Arthur W. Burks ）和 荷曼·哥斯廷 （Herman H. Goldstine）共同执笔的题为"电子计算器件逻辑设计的初步分析及讨论（*Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*）"的论文中，他描述了几个 EDVAC 比 ENIAC 更加先进的特点。EDVAC 的设计者们感觉到计算机内部中应当使用二进制数，而 ENIAC 使用的是十进制数。同时他们认为计算机中应当拥有尽可能大容量的存储器，这些存储器应该用来存储程序代码和程序执行中产生的数据（再说明一下，这些在 ENIAC 中都是不能实现的，对于 ENIAC 来说，编程不过是扳动开关和插拔电线的事情）。这些指令在存储器中是顺序存放的，而且可以由程序计数器进行寻址，但允许条件跳转。这就是著名的"存储程序概念"（stored-program concept）。

这些设计上的决策是计算机历史中非常重要的一个进化阶段，现在我们称之为"冯·诺依曼结构"。上一章中设计的计算机就是一个经典冯·诺依曼计算机。但是伴随着冯·诺依曼结构，又出现冯·诺依曼瓶颈（von Neumann bottleneck）。在冯·诺依曼计

算机中，为了执行指令通常需要花费大量的时间先将这些指令从存储器中取出来。我们仔细回忆一下，第 17 章中最后设计的计算机需要花费 3/4 的时间用来取指令。

在 EDVAC 的那个时期，考虑到成本效益，用真空管制造大容量存储器是不可行的，因此那时提出了一些临时的替代方案。其中一个成功的方案是“水银延迟线路存储器”（mercury delay line memory），它使用 5 英尺水银真空管，在管子的一端每隔 1μs 向水银发送一个短脉冲，这些短脉冲大约需要 1ms 到达管子的另一端（可以如同检测声波一样检测到这些短脉冲，并折回开始端），因此一个水银管可以存储大约 1024 位的信息。

直到 20 世纪 50 年代中期人们才开发出了“磁芯存储器”（magnetic core memory）。众多的被磁化的小金属环由电线串起来组成了磁芯存储器。每一个小金属环可以存储 1 位信息。磁芯存储器沿用了很长一段时间才被别的技术取代，所以常常会听见老一辈程序员们把存储器的访问过程叫做“访问磁芯”。

20 世纪 40 年代，对计算机本质进行概念化设想的并非只有约翰·冯·诺依曼一人。

克劳德·香农（Claude Shannon，生于 1916 年）是另外一个非常有影响力的思想家。在第 11 章中讨论了他 1938 年的硕士论文，正是这篇文章确定了开关、继电器以及布尔代数之间的关系。在 1948 年为贝尔电话实验室工作期间，香农在 *Bell System Technical Journal* 上发表了一片题为“通信过程中的代数理论”（*A Mathematical Theory of Communication*）的文章，在这篇文章中他不仅将“位”的概念介绍给了世界，更开创了一个新的研究领域，即著名的“信息论”（information theory）。信息论研究的是数字信息在有噪声（这些噪声通常阻止信息的通过）的情况下传输，以及如何弥补因噪声产生的损失。1949 年，他撰写了第一篇关于如何编程可以让计算机下棋的文章，1952 年他设计了一个通过继电器控制的机械鼠，它可以在一个迷宫中记住路径。骑单车、变戏法这些耍宝也使得香农在贝尔实验室声名鹊起。

诺博尔特·韦纳（Norbert Wiener，1894-1964）从哈佛大学获得数学博士学位时只有 18 岁，其撰写的 *Cybernetics, or Control and Communication in the Animal and Machine*（1948 年）一书使他闻名于世。他使用新创词汇“控制论”（cybernetics：源于希腊语舵手）表示人类和动物的生物过程同计算机和机器人的机械原理之间的关系。大众文化中，人们普遍使用 cyber 作为前缀表示与计算机相关的一切，最著名的一个词，数百万台计算机通过因特网相连被称做“cyberspace”（网络空间），这个词源于计算机科幻小说作家威廉·吉

布森（William Gibson）在 1984 年发表的小说 *Neuromancer* 中“cyberpunk”一词。

1948 年，埃克特与莫奇利（Eckert-Mauchly）计算机公司（后来成为雷明顿兰德公司一部分）开始开发第一台商用计算机——通用自动计算机，或者称为 UNIVAC。这台机器于 1951 年完成，此后就被送到了人口普查局。UNIVAC 在网络应用方面的首次亮相在哥伦比亚广播公司，它被用来预测 1952 年的总统选举结果。沃尔特·克朗凯特（Walter Cronkite）将 UNIVAC 称做“electronic brain”（电脑）。同样是在 1952 年，IBM 发布了其第一个商用计算机系统，代号 701。

自此开始了漫长的公司和政府的计算历史。尽管这段历史很有趣，但我们要追踪另一段历史——如何缩减计算机成本和体积以及让其走入寻常百姓家，这开始于 1947 年一个鲜为人知的电子技术突破。

许多年以来，贝尔电话实验室是一个让天才们对他们感兴趣的一切事物进行研究的地方。非常幸运的是，他们其中的一些人对计算机有浓厚的兴趣。上面提到的乔治和香农就是在贝尔实验室工作时对早期的计算机发展做出了重大贡献。后来，在 20 世纪 70 年代，贝尔实验室成为很有影响的计算机操作系统 UNIX 和编程语言 C 的诞生地，我们将在下面章节中介绍。

当 AT&T（美国电话电报公司）正式将科学与技术的研究同其他的业务分割时，贝尔实验室内部结构发生了改变，在 1925 年 1 月 1 日成立了子公司。贝尔公司的最初目的是发展改良电话系统的相关技术，幸运的是在这种非常模糊的目标下可以做很多技术研究，但是对于电话系统而言，一个显而易见的长期目标是通过电线不失真的传播语音信号。

从 1912 年开始，贝尔系统致力于真空管放大器的研究，为了能让电话系统使用真空管，对其进行了相当数量的研究和设计方面的改进。尽管做了大量的工作，真空管仍然有许多必须改进的地方。真空管体积太大、耗能大，并且最终会烧毁，但是，它们在当时是唯一的选择。

1947 年 12 月 16 日，当贝尔实验室的两个物理学家约翰·巴丁（John Bardeen，1908-1991）和沃尔特·布兰坦（Walter Brattain，1902-1987）制作出另一种放大器时，所有的一切都发生了改变。这种新型的放大器用一块锗（一种半导体元素）平板和一条黄金薄片制成。一周之后，他们将这个东西演示给他们的老板威廉·肖克利（William

Shockley，1910-1989）。这就是第一个“晶体管”（transistor），它被一些人称为20世纪最重要的发明之一。

晶体管并不是凭空产生的。因为早在8年前，即1939年12月29日，肖克利在他的笔记本上写道“今天我突然想到，使用半导体来制作放大器从原理上讲比使用真空管更为可能。”在晶体管诞生后的几十年里，人们不断地完善它。1956年肖克利、巴丁和布兰坦“因为他们在半导体上的研究以及晶体管效应的发现”获得了当年的诺贝尔物理学奖。

在本书的开始部分探讨了导体和绝缘体。导体因为它们可以有利于电流的通过而得名。铜、银以及金都是很好的导体，元素周期表中这三种元素同属一列并非巧合。

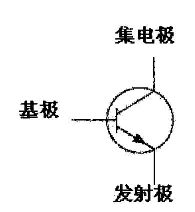
前面谈到过，原子中的电子分布在原子核外，并围绕原子核运动。这三种导体的共同特征是在原子核最外层都有一个单独的电子，而这个电子可以很容易地与原子中的其他电子剥离，因此可以自由移动形成电流。与导体对应的是绝缘体——比如橡胶和塑料——几乎不能导电。

锗元素和硅元素（以及一些化合物）被称为“半导体”（semiconductor），之所以称为“半导体”并不是因为它们的导电性能是导体的一半，而是因为它们的导电系数可以通过多种方式操控。半导体的原子核在最外层有 4 个电子，是外层所能拥有的最大电子数目的一半。纯半导体中，原子之间形成稳定的化学键以及类似金刚石的结构。这种半导体不是良好的导体。

但是，半导体可以掺入一些杂质，即与某些杂质组合。一种类型的杂质称做N型（N表示negative）半导体，它们为原子之间的结合提供额外的电子。另外一种类型的杂质被称做P型半导体。

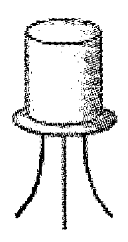
把一个P型半导体夹在两个N型半导体之间可以使之成为一个放大器。这就是著名的NPN晶体管，其三部分分别为集电极（collector）、基极（base），以及发射极（emitter）。

下面是NPN晶体管原理示意图。



在基极施加微小的电压就可以控制非常大的电压从集电极到发射极。如果在基极没有施加电压，那么晶体管将不起作用。

晶体管通常封装在直径为四分之一英寸的小金属罐中，并伸出三根金属线，外形如下图所示。



晶体管开创了固态电子器件的时代，即指晶体管不再需要真空而是使用固体制造，尤其是使用半导体以及当今最为常见的硅来制造。除了体积比真空管更小，晶体管需要的电量更小，产生的热量更少，而且持久耐用。随身携带一个真空管收音机是无法想象的一件事情。但晶体管收音机不同，它可以由一节电池供电，而且不会发烫。1954 年，对于一些幸运的人来说，或许在圣诞节早上打开礼物盒时能获得一件可以随身携带的晶体管收音机。德州仪器公司，半导体革命中一个非常重要的公司，制造了第一批可以随身携带的晶体管收音机。

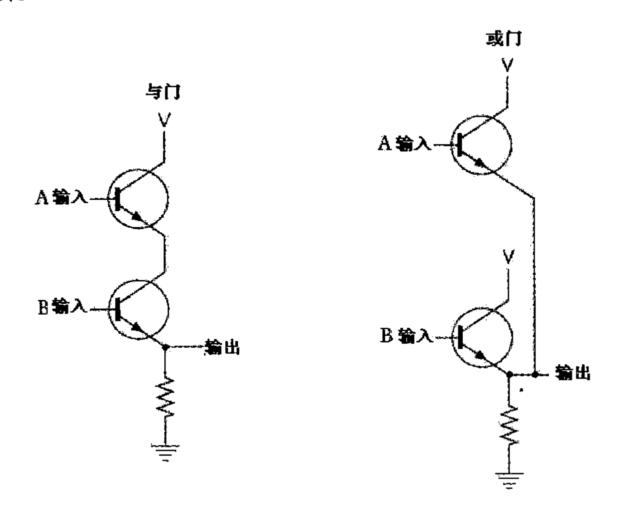
可是，晶体管真正的商业应用却始于助听器。为了纪念亚历山大·格雷厄姆·贝尔（Alexander Graham Bell）为聋人奉献毕生精力，AT&T 公司允许助听器制造商无偿使用晶体管技术。晶体管电视机诞生于 1960 年，到现在电子管的应用几乎已经消失了（可是，并非完全消失，一些高保真音响爱好者以及电子吉他弹奏者较热衷于电子管设备，他们更喜欢真空管放大器产生的音质）。

1956 年，肖克利离开了贝尔实验室成立了肖克利半导体实验室（Shockley Semiconductor Laboratories）。他回到了自己出生的地方，加利福尼亚帕罗奥图市。他的公司是第一个落户于该地区的大公司。其他的半导体和计算机公司立刻也在该地区建立基业，旧金山南部的这个地区现在被人称为硅谷（Silicon Valley）。

开发真空管的最初目的是为了放大电信号，但是它们同样可以应用在逻辑门的开关上，作用与晶体管一样。下面你将看到非常类似于继电器形式的由晶体管构造的与门。

只有当 A 和 B 输入同时为 1 时晶体管才可以导通电流，从而输出为 1。电阻的作用是预防短路。

按照下图右边的方式连接两个晶体管可以组成一个或门。在与门中，上端晶体管的发射极连接下端晶体管的集电极。在或门中，两个晶体管的集电极都与电压源连接，两个发射极相互连接。



使用继电器构造逻辑门以及其他的部件的方法对于晶体管同样是有效的。继电器、真空管以及晶体管最初都是为了开发放大器设计的，但是通过相似方式连接可以组成逻辑门，而计算机则是由这些部件构成的。1956 年诞生了第一台晶体管计算机，随后的几年里，在新型计算机设计中电子管就被淘汰了。

有一个疑问：晶体管肯定可以使计算机更加可靠、体积更小以及需要的电量更少，但晶体管可以使计算机的结构变得更简单么？

答案是否定的。晶体管允许在更小的空间里安装更多的逻辑门，但是你还是要考虑这些组件之间的互连问题。把晶体管连接起来构造逻辑门，与把继电器和真空管连接起来构造逻辑门一样困难。在某些方面来看，这更加困难，因为晶体管更加小而且不容易被控制。如果你想用晶体管制造第 17 章中的计算机和 64 KB 的 RAM，设计工作的重要部分应当是构造某种可以放置所有部件的结构。而你的大部分的体力劳动是在数百万只晶体管之中连接数百万根线，这是很乏味的。

可是，我们已经发现晶体管的某些组合具有特定功能，可以重复利用。一对晶体管可以连接成门，而门常常可以连接成振荡器、加法器、选择器，以及解码器。振荡器可以组成多位锁存器或者 RAM 阵列。如果把晶体管预先连接成常见的构件，再用其来组装计算机会更加容易。

这种设想由英国物理学家杰里佛（Geoffrey Dummer，生于 1909 年）在 1952 年 5 月的一次演讲中提出，他说："我希望展望未来，"，接下来他提出了以下观点：

> "随着晶体管的出现以及半导体研究的广泛开展，现在也许可以设想将来会出现不采用连线而是由固体块组成的电子设备。这种固体块可能由绝缘层、导体层、整流层以及放大层四个层次组成，将不同层次的隔离区连接起来即可实现电子功能。"

然而，真正可以使用的产品还需要再等上几年。

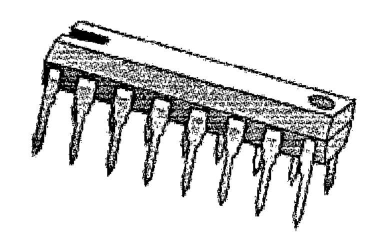
1958 年 7 月，德州仪器公司的杰克·基尔比（Jack Kilby，生于 1923 年）想到了一个可以在一块硅片制造出多个晶体管、电阻和其他电子元件的方法，而他并不知道杰里佛预言。6 个月过后，也就是 1959 年 1 月，罗伯特·诺依斯（Robert Noyce，1927-1990）也想到了类似的方法。诺依斯起初是为肖克利半导体实验室工作，但在 1957 年，他与其他 7 位科学家离开了肖克利半导体实验室创办了仙童（Fairchild）半导体公司。

在技术的发展史中，同时产生一项发明是较常见的，这可能超出了人们的想象。尽管基尔比比诺依斯早 6 个月发明了这种设备，而且德州仪器公司先于仙童公司申请专利，但却是诺依斯首先获得了专利。因此产生了法律上的纠纷，但过了 10 年后，问题才得到令双方都满意的解决。尽管基尔比和诺依斯并没有在一起共事，但今天他们俩被称为集成电路，或者叫做 IC（更通俗的说法是芯片）的共同发明者。

集成电路需要经过非常复杂的工艺流程才可以制造出来，包括将硅片分层，然后非常精确地掺入杂质以及蚀刻不同的区域形成微小组件。开发一种新的集成电路尽管很昂贵，但可以大量生产中获得效益——产量越大，价格就越便宜。

实际上，硅片是薄而且易碎的，因此它必须被安全地封装起来，这样不仅可以起到保护作用，还可以为芯片内部的部件与其他芯片之间的连接提供某种便利。集成电路有几种不同的封装方式，但最为常见的是采用矩形塑料双排直插式（或称为 DIP），提供 14、

16或者40个管脚。



上图是一个有16个管脚的芯片。将芯片上的凹槽朝左放置（如图），用1到16对管脚进行编号，从左下角开始，环绕到右端，依次为1~16，16号管脚位于左边最上端。管脚之间的距离正好是1/10英寸。

纵观20世纪60年代，太空项目以及军备竞赛推动了早期的集成电路市场的发展。在民用方面，第一台用集成电路构造的商品是极点公司（Zenith）在1964年出售的助听器。1971年，德州仪器公司开始出售第一批便携计算器，同年，脉冲星公司（Pulsar）出售了第一块电子手表（电子手表中的集成电路当然不是刚才图示的例子那样的）。随后其他利用了集成电路的产品陆续出现。

1965年，戈登·E·摩尔（Gordon E. Moore，当时在仙童公司工作，后来成为英特尔公司的合伙创办人）发现从1959年以后，技术在以这样一种方式发展：同一块芯片上可以集成的晶体管的数目每年翻一倍。他预测这种趋势将会持续。真实的发展速度比摩尔的发现稍慢一些，因此摩尔定律（最终命名）被修正为：每18个月同一块芯片上集成晶体管数目就会翻一倍。这仍是一个令人吃惊的速度，它解释了为什么刚刚过了几年就家用计算机好像已经过时了。一些人相信直到2015年摩尔定律仍然有效。

发展的早期，人们常常谈论小规模集成电路（small-scale integration），即SSI，指那些逻辑门小少10个的芯片；中规模集成电路（medium-scale integration），即MSI（包含10到100个逻辑门）；大规模集成电路（large-scale integration），即LSI（包含100到5000个逻辑门）。随后的术语为特大规模集成电路（very-large-scale integration），即VLSI（包含5000到50,000个逻辑门）；超大规模集成电路（super-large-scale integration），即SLSI（包含50,000到100,000个逻辑门）；超特大规模集成电路（ultra-large-scale integration，超过100,000个逻辑门）。

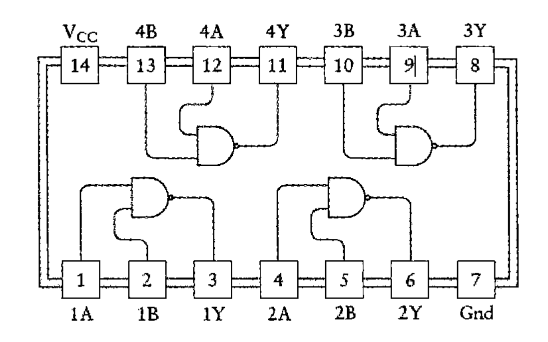
本章的剩余部分以及下一章，我想将时间停留在20世纪70年代中期，此时正是第

一部《星球大战》电影发行前的时代，而 VLSI 处于萌芽阶段。那时，人们使用几种不同的技术来制造集成电路的组件。有时每一种技术被称之为一个 IC 家族，到 20 世纪 70 年代中期，有两个“家族”盛行开来：TTL 和 CMOS。

TTL 代表 transistor-transistor logic（晶体管-晶体管逻辑）。20 世纪 70 年代中期，如果你身为一名数字电路设计师（用 IC 设计大规模电路），那么一本 1.25 英寸厚、由德州仪器公司在 1973 年出版的名为 *The TTL Data Book for Design Engineers*（《TTL 工程师设计数据手册》，以下简称《TTL 数据手册》的书将会是你书桌上的常客。这是一本德州仪器和其他几个公司出售的 TTL 集成电路 7400 系列完整的参考书，由之所以这样称呼是因为这个 IC“家族”的每一名“成员”都是以数字 74 开头。

7400 系列中的每一个集成电路都是由以特定方式连接的预留逻辑门组成。一些芯片提供简单的预留的逻辑门，设计者可以用它们来组成更大规模的组件；另外一些芯片则提供通用组件，例如：触发器、加法器、选择器以及解码器。

7400 系列中第一个集成电路标号即为 7400，在《TTL 数据手册》中这样描述它——“四个双输入正与非门”。这意味着这个特殊的集成电路包含四个双输入与非门。“正”与门则是指 1 对应为有电压，而 0 对应为没有电压。下图是一个 14 管脚的芯片，数据手册中的一张小图显示了管脚对应的输入与输出。



上面这张图为芯片的俯视图（管脚在下面），小凹槽位于左边。

14 号管脚标注为 V_{CC}，与符号 V 一样，用来代表电压（顺便说一下，大写字母 V 的双下标代表电压源。下标的字母 C 指晶体管的电压输入端，即集电极，collector）。7 号管脚标注的 GND 代表接地（ground）。在特定电路中使用的所有集成电路都必须有接电源端与接地端。

拿 TTL7400 系列来说，V_{CC} 值必须介于 4.75V 和 5.25V 之间。换句话讲，电压必须在 5V ± 5% 的范围。电压低于 4.75V 时，芯片将无法工作。而高于 5.25V 时，芯片将被烧坏。即便有一个 5V 的电池，那也不能用来对 TTL 进行供电，因为电池的电压不可能刚好适合这些芯片。通常情况下，TTL 需要从墙上接入电源。

7400 芯片中每一个与非门有两个输入端和一个输出端，且相互独立工作。上一章中已经区分了输入为 1（有电压）或者为 0（无电压）的情况。实际上，与非门的输入端电压可以为 0V（接地）到 5V（V_{CC}）范围内的任一值。TTL 中，当电压介于 0 ~ 0.8V 任一值则可以认为是逻辑“0”，而介于 2 ~ 5V 则可以认为是逻辑“1”。0.8 ~ 2V 范围的电压输入则应当尽量避免。

TTL 的典型输出是以 0.2V 表示逻辑“0”，以 3.4V 表示逻辑“1”。考虑到电压值不稳定，有时会有一些波动，集成电路的输入和输出端有时不用“0”和“1”表示，而是用“低”和“高”表示。此外，有时候低电压可以表示逻辑“1”，而高电压则可以表示逻辑“0”，这种配置称为“负逻辑”。7400 芯片被称为“四个双输入正与非门”，而这里的“正”则代表了上述所讲的正逻辑。

如果 TTL 的典型输出 0.2 V 代表逻辑“0”，而 3.4 V 代表逻辑“1”，那么这个输出电压确实是在输入允许的范围内，即逻辑“0”为 0 ~ 0 . 8 V，且逻辑“1”为 2 ~ 5 V，这就是 TTL 可以隔离噪声的原因。逻辑“1”输出的电压哪怕下降 1.4V 也仍可以作为逻辑“1”的高电压输入，同样的，逻辑“0”输出的电压哪怕升高 0.6V 也仍可以作为逻辑“0”的低电压输入。

影响一个集成电路性能的最重要因素可以认为是传播时间（propagation time），也就是输入端发生变化引起输出端发生相应变化所需要的时间。

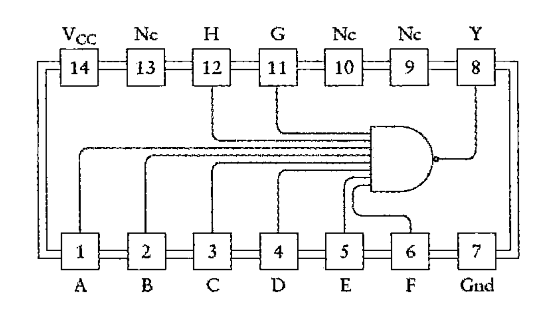
通常以纳秒来衡量芯片的传播时间，缩写为 nsec，即 ns。1 纳秒是非常短的时间。千分之一秒称为毫秒，百万分之一秒称之为微秒，那么十亿分之一秒则称为纳秒。7400 芯片中与非门的传播时间应该保证小于 22 ns，即 0.000000022s。

感觉不到纳秒长短的并非只有你一人。地球上的所有人对纳秒只有概念上的理解，除此之外别无所有。纳秒比人类感觉到的任何事情都要短暂得多，以至于永远无法理解它。任何解释只会将纳秒变得更加难以捉摸。比如，当你拿着一本离你脸部距离为 1 英

尺的书时，那么纳秒可以定义为光从书页到你的眼睛的时间，但这种解释可以使你更好地认识纳秒么?

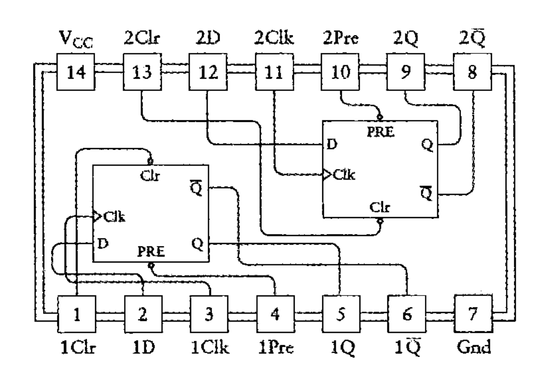
然而，纳秒使计算机成为可能。正如在第 17 章中看到的，计算机处理器迟钝地做着简单的事情——从存储器中取出一个字节放到寄存器中，再将两个字节相加，然后将结果存放回存储器。迅速地完成这些操作是计算机（并非第 17 章中的计算机，而是当今使用的计算机）能完成任何实际工作的唯一原因。诺依斯说过："当更好地认识了纳秒后，从概念上来讲计算机操作是相当简单的"。

继续阅读《TTL 数据手册》会发现书中很多熟悉的小条目。7402 芯片有 4 个双输入或非门，7404 芯片有 6 个反相器，7408 芯片有 4 个双输入与门，7432 芯片有 4 个双输入或门，以及 7430 芯片有一个 8 输入与非门，如下图所示。

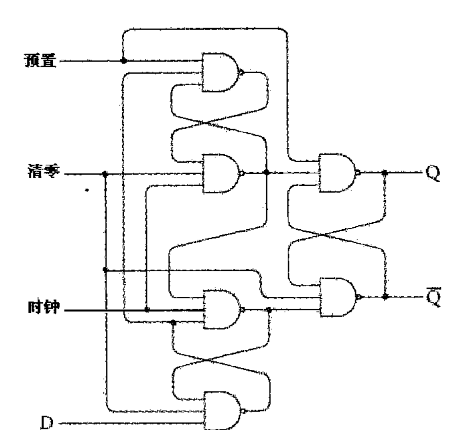


缩写 Nc 表示无连接（no connection）。

7474 芯片是另一个较为熟悉的芯片。它是一个"带预置和清零的双D型正边沿触发器"，如下图所示。



《TTL 数据手册》中甚至还囊括了这个芯片中每个触发器的逻辑图。



除了使用的是异或门外，你会发现上面的这个图与第 14 章结尾的图很相似。《TTL 数据手册》中的逻辑表也稍微不同。

输入				输出	
Pre	Clr	Clk	D	Q	$\overline{Q}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H*	H*
H	L	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	Q_0	$\overline{Q}_0$

表格中，“H”代表高电平，“L”代表低电平。当然，可以将这些想象成 1 和 0。在触发器中，预置（Pre）和清零输入（Clr）通常为 0；这里通常为 1。

继续阅读《TTL 数据手册》会发现，7483 芯片是一个 4 位二进制全加法器，74151 是一个 8-1 的数据选择器，74154 芯片是一个 4-16 的解码器，74161 芯片是一个 4 位同步二进制计数器，以及 74175 芯片是一个带清零的 4 输入 D 型触发器。从上述芯片中挑出两种可以制作一个 8 位锁存器。

现在你应该知道我是怎么想出从第 11 章中就开始使用的这么多不同种类的组件的，这些都是从《TTL 数据手册》借鉴而来。

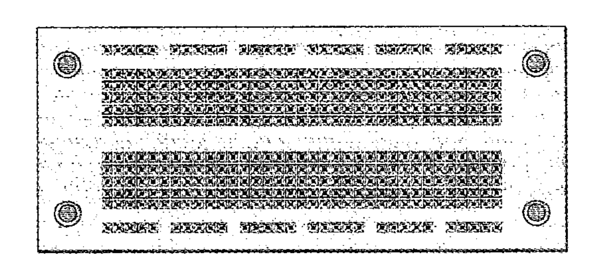
作为一名数字电路设计工程师，你应当多花点时间看完《TTL 数据手册》这本书，使自己熟悉用得到的 TTL 芯片类型。一旦你熟知了使用的工具，那么就可以使用 TTL 芯片构造一台第 17 章中的计算机。将芯片连接起来比将一个个的晶体管连接起来容易得多，但你可能不会考虑使用 TTL 来做 64KB RAM 阵列。1973 年出版的《TTL 数据手册》中，列出的容量最大的 RAM 芯片仅仅只有 256×1 位，制造 64 KB 需要 2048 个芯片！对制造存储器来讲，TTL 绝非最好的技术。关于存储器将在第 21 章中详细讨论。

或许你想使用更好一点的振荡器。只要将 TTL 反相器的输出连接到输入，就会获得一个振荡器，而且其振荡频率更容易计算。这种振荡器使用石英晶体制造相当简单，石英晶体放在带有两个引线的密封小扁罐中。这些石英晶体的振荡频率在一个特定的值，通常情况下是每秒至少振荡一百万个周期，称 1 兆赫兹，缩写为 MHz。如果要使用 TTL 制造第 17 章中的计算机，那么需要时钟频率为 10 MHz 才可以使其运行良好，每条指令执行时间为 400 ns。当然，这比使用继电器来做任何我们所构想的事情都要快。

芯片家族中另一位明星（至今仍是）是 CMOS，CMOS 表示互补金属氧化物半导体（complementary metal-oxide semiconductor）。如果你是一名 20 世纪 70 年代中期的使用 CMOS 集成电路进行电路设计的爱好者，那么可能会使用到一本名为《CMOS 数据手册》的书作为参考源，该书由美国国家半导体公司出版，可以在当地的 Radio Shack 商店买到，书中涵盖了 CMOS 家族中 4000 系列的 IC 的信息。

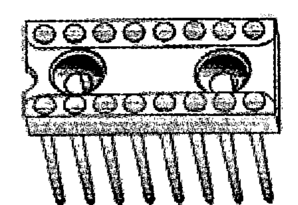
TTL 供电电压要求在 4.75～5.25V 范围内。而对于 CMOS 来说，范围在 3～18V 内的电压均可，这是非常灵活的。此外，CMOS 相比 TTL 需要更少的能量，这使得用电池运行小型 CMOS 电路变得可行。CMOS 的缺点是速度慢，比如，在供电电压为 5 伏的情况下，CMOS 4008 4 位全加器可以保证的传播时间只有 750 ns。CMOS 芯片的传播速度会随着电压提高而提高——10V 时为 250ns，15 伏时为 190 ns，但仍不能与 TTL4 位加法器的 24 ns 的传播时间相媲美（25 年前，TTL 的速度与 CMOS 低功率之间的权衡是非常清晰的，斗转星移，今天 TTL 已经拥有了低功率版本而 CMOS 也有了高速版本）。

在实际应用中，芯片的连接是在一个塑料“面包板”（如下图所示）完成的。



每一短行有 5 个孔，在塑料板背面这 5 个孔通过电线相连。把芯片插入到面包板中，芯片将横跨在中间长槽的两侧，芯片的管脚则分别插到槽两侧的孔里，这样芯片的每一个管脚都会与其他四个孔里的管脚相连接，通过在孔之间连接电线可以实现芯片之间的连接。

使用一种叫做“钢丝包装”（wire-wrapping）的技术可以使芯片之间连接更加牢固，芯片插入到带有几个长长的方柱的插槽中，如下图所示。



每一个柱体对应芯片的一个管脚，而插口本身则插入到事先穿孔的薄板中。在板子的另一面，使用特殊的钢丝包装枪将每一个柱体周围紧紧包上绝缘线。柱体的直角边缘则从绝缘线中破出，与导线相连。

如果实际应用中使用集成电路制造一个特定的电路，那可能会用到“印刷电路板”（printed circuit board）。很久以前，这是集成电路爱好者做的事情。这种电路板上面布满了洞，并且被一层薄铜片覆盖。基本上，可以让防酸物质覆盖铜片上所有你想保护的地方，而使用酸蚀刻剩余的部分，接着就可以将集成电路的插口（或者是集成电路本身）直接焊接到电路板上的铜片上，但由于集成电路中存在很多互相连接，仅覆盖一层铜片通常情况下无法完成电路，所以商业制造的印刷电路板有多层互连。

到 20 世纪 70 年代早期，使用集成电路在一块电路板上制造一个完整的计算机处理器变得可能，实际上这距离将整个处理器放入一块芯片中，只是一个时间问题。虽然德州仪器公司在 1971 年为一块单芯片计算机提交了专利申请，但真正制造出一块这种单片

机芯片的荣誉却属于英特尔公司（英特尔公司成立于 1968 年，由仙童公司以前的雇员罗伯特·诺伊斯和戈登·摩尔合伙创办）。1970 年，英特尔发售了第一款产品，一个可以存储 1024 位数据的芯片，在当时这是单一芯片中可以存储的最大位数。

英特尔在为日本吉康（Busicom）公司生产的可编程计算器设计芯片的过程中，决定采取一种不同的方法。正如英特尔工程师特德·霍夫（Ted Hoff）说的："我想让它成为一个具有通用功能的计算机，进而可以通过编程成为一个计算器，而不是使这个设备成为一个只有一些编程能力的计算器，" 这导致了 Intel 4004 的产生，它是第一块"计算机芯片"，或者叫做"微处理器"。1971 年 11 月，4040 芯片已经可以得到使用，它拥有 2300 个晶体管（依照摩尔定律，18 年后微处理器包含的晶体管数将是这个数字的 4000 倍，或者说是 1000 万。这是一个相当精确的预测）。

我们已经知道 4004 芯片包含的晶体管数目，下面是 4004 芯片另外三种重要的特征。自 4004 芯片开始，在比较微处理器性能时，通常采用三个衡量标准。

第一个标准：4004 是一个 4 位微处理器，这意味着处理器中数据通路宽度只有 4 位。每次做加、减运算时，它只能处理 4 位的数字。对比来看，第 17 章中的计算机数据通路是 8 位，因此被称为 8 位处理器。我们即将看到 8 位处理器很快就超越了 4 位处理器。但技术并没有停止于此，20 世纪 70 年代末期，16 位微处理器已经得到了应用。回想一下第 17 章中的内容，以及在 8 位处理器中进行两个 16 位数加法所必需的指令码，你就会欣喜地发现 16 位处理器带来的优势。到 20 世纪 80 年代中期，32 位微处理器诞生了，并自此一直作为家用计算机的主要处理器。

第二个标准：4004 每秒最大时钟频率为 108,000 周期，即 108 KHz。时钟频率是指连接到微处理器并驱动它运行的振荡器的最大频率，超过此时钟频率，微处理器将不能正常工作。到 1999 年，家用计算机的微处理器已经达到了 500 MHz——比 4004 要快 5000 倍。

第三个标准：4004 的可寻址存储器只有 640 字节，现在来看这个数字小得有点荒唐，但这与当时可得的存储芯片的容量是一致的。下一章中你将会看到，两年之中微处理器的寻址能力就达到了 64 KB，与第 17 章中计算机的能力比肩。1999 年英特尔生产的芯片可以寻址 64 TB 的空间，尽管当时多数人的家用电脑 RAM 容量还不到 256 MB。

上述三个数字指标并不能影响一台计算机的计算能力。比如，4 位处理器同样可以实

现 32 位数字加法，只不过是将其简单拆分为 4 位的数来进行。某种意义上讲，所有的数字计算机都是相同的，如果一台处理器从硬件上无法做到另外一台可以做到的事情，那么它可以通过软件途径做到，最终它们可以完成相同的事情，这是 1937 年图灵在论文里面关于可计算性的一种定义。

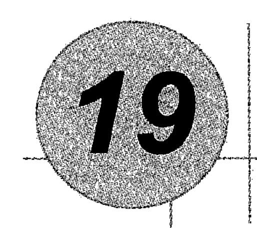
然而，速度是处理器之间的根本不同点，同时速度也是我们使用计算机的一大原因。

最大时钟频率（maximum clock speed），也称为主频，是影响处理器速度的决定性因素之一。时钟频率决定了执行一条指令所需要的时间，处理器的数据位宽也影响处理器的速度。尽管 4 位处理器可以完成 32 位数字的加法，但速度是不能与 32 位处理器相媲美的。然而，令人感到迷惑的是，处理器可寻址存储器的最大空间对处理器速度也是有影响的，首先，可寻址存储器看上去只反映了处理器的某些能力，尤其是在需要大容量存储器的前提下进行数据处理的能力，而与处理器速度无关。但其实，处理器可以利用某些存储器地址去控制其他的介质来存取信息，这样就绕开了存储器容量的限制（比如，假设在特定的存储器地址写入一个字节就在一个纸带上穿一个孔，而从存储器中读出一个字节等于从纸带上读取一个孔一样）。可是这种处理办法会降低整个计算机的速度——这就又回到了速度问题！

当然，这三个数字只能粗略地反映微处理器操作的速度，它们并不能体现出微处理器的内部构造以及机器指令代码的效率和能力。随着处理器变得越来越复杂，以前通过软件完成的任务都可以在处理器上完成，我们将会在后面的章节中看到这方面的例子。

即使所有的计算机具有相同的计算能力，即使它们只能做和图灵设计的早期计算机一样简单的事情，但有一点是无法回避的，那就是处理器的速度最终决定了其用途。比如那些表现比人脑还慢的计算机是毫无用处的，跟进一步来说，如果处理器需要用一分钟来画一帧图像，那么对于现代计算机而言，要想在其屏幕上播放电影也是不可能实现的。

回到 20 世纪 70 年代中期，虽然 4004 有很多局限性，但毕竟只是个开始。到 1972 年 4 月，英特尔发布了 8008 芯片——一个时钟频率为 200 KHz、可寻址空间为 16 KB 的 8 位微处理器（瞧，用三个数字来总结一个处理器是多么简单的事）。后来在 1974 年 5 月，英特尔公司和摩托罗拉公司同时发布了 8008 微处理器的改进版，正是两款芯片改变了整个世界。



两种典型的微处理器

微处理器——正是它，将计算机中央处理器的所有构成组件整合在一起，集成在一个硅芯片上——诞生于 1971 年。它的诞生有着很好的开端：第一个微处理器，即 Intel 4004 系列，包括了 2300 个晶体管。到现在，大约三十年过去了，家用计算机的微处理器中的晶体管数量也逐步逼近 10,000,000 个。

从本质上说，微处理器实际上所做的工作一直没有变。在现在的芯片上，新增的几百万个晶体管所做的很多事情令我们眼前一亮，但我们正处于微处理器探索的初期，过多的关心这些当代的芯片并不合适，因为它们只会分散我们的注意力而无法帮助我们去学习与理解它。为了更清晰地认识微处理器是如何工作的，让我们首先来看一下最原始的微处理器。

我们要讨论的微处理器出现于 1974 年。在这一年，英特尔公司在 4 月推出了 8080 处理器，摩托罗拉公司——从 20 世纪 50 年代生产半导体和晶体管——在 8 月推出了 6800 处理器。不仅如此，当年还有其他的一些微处理器面世。同年，德克萨斯仪器设备公司

（Texas Instruments）推出了 4 位的处理器 TMS 1000，它用于多种计算器、玩具和设备；国家半导体公司（National Semiconductor）推出了 PACE——首个 16 位微处理器。但当我们回顾历史的时候就会发现，8080 和 6800 是两个最具有重大历史意义的芯片。

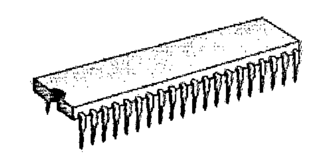
英特尔为 8080 最初定的价格为 360 美元，这个价格对 IBM 的 System/360 来说是极大的讽刺。System/360 是大型机处理系统，用户大都是一些大公司，售价动辄几百万美元（今天，你用 1.95 美元就可以买到一块 8080 芯片）。这并不是说 8080 可以与 System/360 相提并论，但在几年之内，IBM 自己也关注起这些非常小的计算机。

8080 是一个 8 位的微处理器，它包括 6000 个晶体管，运行的时钟频率为 2 MHz，寻址空间为 64 KB。摩托罗拉的 6800（今天的售价也是 1.95 美元）包括 4000 个晶体管，其寻址空间也是 64 KB。第一个版本的 6800 的运行速度为 1 MHz，但摩托罗拉于 1977 年推出了运行速度分别为 1.5 MHz 和 2 MHz 的版本。

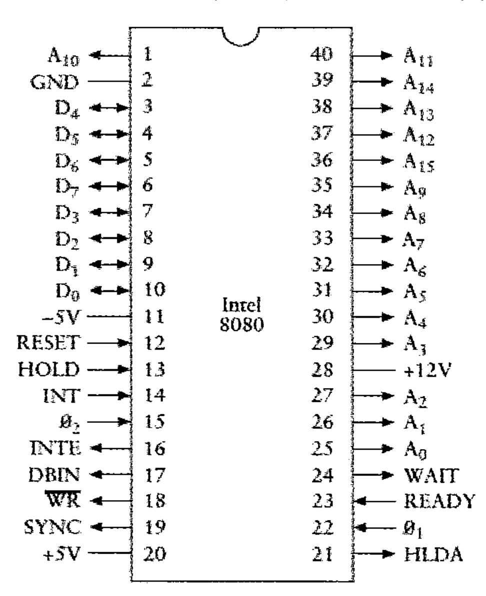
这些芯片被称为"单芯片微处理器"（single-chip microprocessors），不太准确的说法是"单芯片的计算机"。处理器只是计算机的一部分。除了处理器之外，计算机还需要其他一些设备，至少要包括一些随机访问的存储器（RAM），一些方便用户把信息输入计算机的设备（输入设备），一些使用户能够把信息从计算机中读取出来的设备（输出设备），以及其他一些能把所有构件连接在一块的芯片。本书会在第 21 章详细介绍这些构件。

现在，让我们来仔细研究一下微处理器本身。当描述微处理器的时候，我们总是习惯用一些框图来阐明其内部的构件及其连接情况。然而，在第 17 章我们已经使用了数不清的框图来描述它，现在我们将观察微处理器和外部设备的交互过程，以此来认识其内部的结构和工作原理。换句话说，为了弄清微处理器的工作原理，我们把它视做一个不需要详细研究其内部操作的黑盒。取而代之的方法是通过观测芯片的输入、输出信号，特别是芯片的指令集来理解微处理器的工作原理。

8080 和 6800 都是 40 个管脚的集成电路。这些芯片最常见的 IC 封装大约为 2 英寸长，1/2 英寸宽，1/8 英寸厚。



当然，你所看到的只是外部的封装。其内部的硅晶片是非常小的，例如在早期的 8 位微处理器中，硅晶片还不到 1/4 平方英寸。外包装可以保护内部的硅晶片，并且通过管脚提供了处理器的输入和输出访问接入点。下面给出了 8080 的 40 个管脚的功能说明图。



本书中我们所创建的所有电气或电子设备都需要某种电源来供电。8080 的一个特殊的地方就是它需要三种电源电压：管脚 20 必须接到 5V 的电压；管脚 11 需要接到-5V 的电压；管脚 28 需接 12V 的电压；管脚 2 接地。（英特尔在 1976 年发布了 8085 芯片，目的就是简化对这些电源的要求）

其他的管脚都标有箭头。从芯片引出的箭头表明这是一个输出（output）信号，这种信号由微处理器控制，计算机的其他芯片对该信号响应。指向芯片的箭头表明该信号是一个输入（input）信号，该信号由其他芯片发出，并由 8080 芯片对其响应。还一些管脚既是输入又是输出。

第 17 章所设计的处理器需要一个振荡器来使其工作。8080 需要两个不同的同步时钟输入，它们的频率都是 2 MHz，分别标记为 ϕ_1 和 ϕ_2，位于管脚 22 和 15 上。这些信号可以很方便地由英特尔生产的 8224 时钟信号发生器产生。为 8224 连接一个 18 MHz 的石英晶体后，它基本上就可以完成其余工作了。

一个微处理器通常有多个用来寻址存储器的输出信号。用于寻址的输出信号的数目

与微处理器的可寻址空间大小直接相关。8080 有 16 个用于寻址的输出信号，标记为 A_0 ~ A_{15}，因此它的可寻址空间大小为 2^{16}，即 65,536 字节。

8080 是一个 8 位的微处理器，可以一次从存储器读取或向存储器写入 8 位数据。该芯片还包括标记为 D_0 ~ D_7 的 8 个信号，这些信号是芯片仅有的几个既可以用做输入又可以用做输出的信号。当微处理器从存储器中读取一个字节时，这些管脚的功能是输入；当微处理器向存储器写入一个字节时，其功能又变成了输出。

芯片的其余 10 个管脚是控制信号（control signals）。例如，RESET（复位）输入用于控制微处理器的复位。输出信号 $\overline{WR}$ 的功能是指明微处理器需要向 RAM 中写入数据（$\overline{WR}$ 信号对应于 RAM 阵列的写输入）。此外，当芯片读取指令时，在某些时刻一些控制信号会出现在 D_0 ~ D_7 管脚处。使用 8080 芯片构建的计算机系统通常使用 8228 系统控制芯片来锁存附加的控制信号。本章在后面将会讲述一些控制信号。但 8080 的控制信号是极其复杂的，因此，除非你准备用该芯片搭建一台计算机，否则最好不要在这些控制信号上过多花费时间。

假设 8080 微处理器连接了一个 64KB 的存储器，这样我们就能独立地读写数据而不依赖于微处理器。

8080 芯片复位后，它把锁存在存储器 0000h 地址处的字节读入微处理器，通过在地址信号端 A_0 ~ A_{15} 输出 16 个 0 实现该过程。它读取的字节必须是 8080 指令，读取该字节的过程被称为取指令（instruction fetch）。

在第 17 章设计的计算机中，所有的指令（除了 HLT 指令）都是 3 个字节长，包括 1 字节的操作码和 2 字节的地址。在 8080 中，指令的长度可以是 1 字节、2 字节，或者 3 字节。有些指令使 8080 从存储器的一个特定地址读取字节到微处理器，有些指令使 8080 将一个字节从微处理器写入存储器的特定地址；还有些指令使 8080 在其内部执行而不需要访问 RAM。8080 执行完第一条指令后，接着从存储器读取第二条指令，并依此类推。这些指令组合在一起构成了计算机程序，可以用来做一些很有趣的事情。

当 8080 以最高速度 2 MHz 运行时，每个时钟周期是 500ns（1 ÷ 2,000,000 = 0.000000500s）。第 17 章中的计算机的所有指令都需要 4 个时钟周期，8080 的每条指令需要 4 ~ 18 个时钟周期，这就意味着每条指令的执行时间为 2 ~ 9μs。

也许了解某个特定微处理器的功能的最好办法就是全面地测试其完整的指令集。

第 17 章最后完成的计算机仅包括 12 条指令。一个 8 位处理器的指令数很容易达到 256，每一条指令的操作码就是一个特定的 8 位数（如果某些指令包含 2 字节的操作码，其指令集会更大）。8080 虽然没有这么多指令，但是其指令数也已经达到了 244。这看起来似乎是很多，但从总体上说，其功能并不比第 17 章的计算机强大。例如，如果想利用 8080 进行乘法或除法运算，你仍然需要自己写一小段代码。

在第 17 章曾经讲到过，为了方便地引用指令，我们为处理器的每一条指令的操作码都指派了一个特殊的助记符，而且其中的一些助记符是可以带有参数的。这种助记符只是在我们使用操作码时提供方便，它对于处理器是没有帮助的，处理器只能读取字节，对于助记符组成的文本的含义一无所知（为了讲解清楚，本书选用了 Intel 8080 说明文档中用到的部分助记符为例来说明）。

第 17 章设计的计算机的指令集包括两条非常重要的指令，我们称之为加载（Load）和保存（Store）。每条指令占 3 个字节。在 Load 指令中，第一个字节是操作码，其后的两个字节是要加载的操作数的 16 位地址。当处理器执行加载指令时，会把该指定地址中的字节加载到累加器。与之相似，当 Store 指令被执行时，累加器中的内容被保存到该指令指定的地址中。

我们可以用助记符把上述代码简写为以下形式：

```
LOD A, [aaaa]
STO [aaaa], A
```

这里的 A 表示累加器（它既是 Load 指令的目的操作数也是 Store 指令的源操作数），aaaa 表示 16 位的存储器地址，通常用 4 个 16 进制的数来表示一个地址。

同第 17 章的累加器一样，8080 的 8 位累加器也记做 A。8080 也有与第 17 章的计算机的 Load 指令和 Store 指令功能相同的两条指令，它们也称做加载（Load）和保存（Store）。在 8080 中，加载指令和保存指令的操作码分别是 32h 和 3Ah，每个操作后面也同样跟着一个 16 位的地址。在 8080 中，它们的助记符分别是 STA（Store Accumulator，表示加载到累加器）和 LDA（Load Accumulator，表示保存到累加器）：

操作码	指　令
32	STA　[aaaa], A
3A	LDA　A, [aaaa]

8080 芯片的微处理器的内部除累加器外还设置了 6 个寄存器（register），每个寄存器可以存放一个 8 位的数。这些寄存器和累加器非常相似，事实上累加器被视为一种特殊的寄存器。这 6 个寄存器和累加器一样，本质上都是锁存器。处理器既可以把数据从存储器读入寄存器，也可以把数据从寄存器存回存储器。当然，其他的寄存器没有累加器所具有的丰富的功能，例如，当把两个 8 位数相加时，其结果总是保存到累加器而不会保存到其他寄存器。

在 8080 中用 B，C，D，E，H 和 L 来表示新增的 6 个寄存器。人们通常会问以下两个问题："为什么不使用 F 和 G 来表示？"，以及"I，J 和 K 用来代表什么？"答案是，使用 H 和 L 来命名寄存器是因为它们具有特殊的含义，H 可以代表高（High）而 L 可以代表低（Low）。通常把两个 8 位的寄存器 H 和 L 合起来构成一个 16 位的寄存器对（register pair），称做 HL，H 用来保存高字节而 L 用来保存低字节。这个 16 位的值通常用来对存储器寻址，我们将在下面看到它是怎样以简单的方式工作的。

寄存器是计算机必不可少的部件吗？为什么在第 17 章搭建的计算机中并没有寄存器的踪迹？从理论上讲，这些寄存器不是必需的，在第 17 章也没有用到它们，但在实际应用中使用它们将带来很大的方便。很多计算机程序都同时用到多个数据，将这些数据存放在寄存器比存放在存储器更便于访问，因为程序访问内存的次数越少其执行速度就越快。

在 8080 中有一条指令至少用到了 63 个操作码，这条指令就是 MOV，即 Move 的缩写。其实该指令是一条单字节指令，它主要用来把一个寄存器中的内容转移到另一个寄存器（也可能就是原来的寄存器）。因为 8080 微处理器设计了 7 个寄存器（包括累加器在内），因此应用中使用大量的 MOV 指令是很正常的。

下面列出了前 32 条 MOV 指令。再一次提醒你，两个参数中左侧的是目标操作数，右侧的是源操作数。

操作码	指　令	操作码	指　令
40	MOV　B, B	50	MOV　D, B
41	MOV　B, C	51	MOV　D, C
42	MOV　B, D	52	MOV　D, D
43	MOV　B, E	53	MOV　D, E
44	MOV　B, H	54	MOV　D, H
45	MOV　B, L	55	MOV　D, L
46	MOV　B, [HL]	56	MOV　D, [HL]
47	MOV　B, A	57	MOV　D, A
48	MOV　C, B	58	MOV　E, B
49	MOV　C, C	59	MOV　E, C
4A	MOV　C, D	5A	MOV　E, D
4B	MOV　C, E	5B	MOV　E, E
4C	MOV　C, H	5C	MOV　E, H
4D	MOV　C, L	5D	MOV　E, L
4E	MOV　C, [HL]	5E	MOV　E, [HL]
4F	MOV　C, A	5F	MOV　E, A

这些指令使用起来非常方便。利用上面的指令可以方便地把一个寄存器存放的数据转移到另一个寄存器。下面让我们研究一下以 HL 寄存器对作为操作数的 4 条指令。

```
MOV B, [HL]
```

前面讲过 LDA 指令，它可以把单字节的操作数从存储器转移到累加器；LDA 操作码后面直接跟着该操作数的 16 位地址。在上面列出的指令中，MOV 指令把字节从存储器转移到 B 寄存器，但该字节的 16 位地址却存放在 HL 寄存器对中。HL 是怎样得到 16 位存储器地址的呢？这并不难解决，有很多方法可以做到，比如通过某种计算实现。

总而言之，对于下面这两条指令：

```
LDA A, [aaaa]
MOV B, [HL]
```

它们的功能都是把一个字节从内存读入微处理器，但它们寻址存储器的方式并不相同。第一种方式称做直接寻址（direct addressing）；第二种方式称做间接寻址（indexed addressing）。

下面列出了其余 32 条 MOV 指令，我们看到 HL 保存的 16 位存储器地址也可以作为

目标操作数。

操作码	指　令	操作码	指　令
40	MOV　B, B	50	MOV　D, B
60	MOV　H, B	70	MOV　[HL], B
61	MOV　H, C	71	MOV　[HL], C
62	MOV　H, D	72	MOV　[HL], D
63	MOV　H, E	73	MOV　[HL], E
64	MOV　H, H	74	MOV　[HL], H
65	MOV　H, L	75	MOV　[HL], L
66	MOV　H, [HL]	76	HLT
67	MOV　H, A	77	MOV　[HL], A
68	MOV　L, B	78	MOV　A, B
69	MOV　L, C	79	MOV　A, C
6A	MOV　L, D	7A	MOV　A, D
6B	MOV　L, E	7B	MOV　A, E
6C	MOV　L, H	7C	MOV　A, H
6D	MOV　L, L	7D	MOV　A, L
6E	MOV　L, [HL]	7E	MOV　A, [HL]
6F	MOV　L, A	7F	MOV　A, A

其中的一些指令如：

```
MOV  A, A
```

并不会执行有意义的操作。而指令：

```
MOV  [HL], [HL]
```

是不存在的，事实上，与之对应的指令是 HLT（Halt）即停止指令，也就是说该指令的意义是停止。

研究 MOV 操作码的位模式能更好地了解它，MOV 操作码由 8 位组成：

```
01dddsss
```

其中 ddd 这 3 位是目标操作数的代码，sss 这 3 位是源操作数的代码。它们所表示的意义如下：

000 = 寄存器 B

001 = 寄存器 C

010 = 寄存器 D

011 = 寄存器 E

100 = 寄存器 H

101 = 寄存器 L

110 = 寄存器 HL 保存的存储器地址中的内容

111 = 累加器 A

例如，指令

```
MOV  L, E
```

对应的操作码为：

```
01101011
```

用十六进制数可表示为 6Bh。这与前面列出的表格是一致的。

可以设想一下，在 8080 的内部可能是这样的：标记为 sss 的 3 位用于 8-1 数据选择器，标记为 ddd 的 3 位用来控制 3-8 译码器以此确定哪一个寄存器锁存了值。

寄存器 B 和 C 也可以组合成 16 位的寄存器对 BC，同样我们还可以用 D 和 E 组成寄存器对 DE。如果这些寄存器对也包含要读取或保存的字节的存储器地址，则可以用下面的指令实现：

操作码	指　令	操作码	指　令
02	STAX [BC], A	0A	LDAX A, [BC]
12	STAX [DE], A	1A	LDAX A, [DE]

另一种类型的传送（Move）指令称做传送立即数（Move Immediate），它的助记符写做 MVI。传送立即数指令是一个双字节指令，第一个字节为操作码，第二个是数据。这个单字节数据从存储器转移到某个寄存器，或者转移到存储器中的某个存储单元，该存储单元由 HL 寄存器对寻址。

操作码	指　令
06	MVI　B, xx
0E	MVI　C, xx
16	MVI　D, xx
1E	MVI　E, xx
26	MVI　H, xx
2E	MVI　L, xx
36	MVI　[HL], xx
3E	MVI　A, xx

例如，当指令：

```
MVI     E, 37h
```

执行后，寄存器 E 存放的字节是 37h。这就是我们要介绍的第三种寻址方式——立即数寻址（immediate addressing）。

下面将列出一个操作码集，包括 32 个操作码，它们能完成 4 种基本的算术运算，这些运算在第 17 章设计处理器时我们已经熟悉了，它们是加法（ADD）、进位加法（ADC）、减法（SUB）和借位减法（SBB）。可以看到，在所有的例子中，累加器始终用于存放其中的一个操作数，同时用来保存计算结果。这些指令如下。

操作码	指　令	操作码	指　令
80	ADD　A, B	90	SUB　A, B
81	ADD　A, C	91	SUB　A, C
82	ADD　A, D	92	SUB　A, D
83	ADD　A, E	93	SUB　A, E
84	ADD　A, H	94	SUB　A, H
85	ADD　A, L	95	SUB　A, L
86	ADD　A, [HL]	96	SUB　A, [HL]
87	ADD　A, A	97	SUB　A, A
88	ADC　A, B	98	SBB　A, B
89	ADC　A, C	99	SBB　A, C
8A	ADC　A, D	9A	SBB　A, D
8B	ADC　A, E	9B	SBB　A, E
8C	ADC　A, H	9C	SBB　A, H
8D	ADC　A, L	9D	SBB　A, L

8E	ADC A, [HL]	9E	SBB A, [HL]
8F	ADC A, A	9F	SBB A, A

假如累加器 A 存放的字节是 35h，累加器 B 存放的字节是 22h，经过减法运算：

```
SUB     A, B
```

累加器中的值变为 22h，即两个字节的差。

如果累加器 A 中的值为 35h，寄存器 H 和 L 中的值分别是 10h 和 7Ch，而存储器地址 107Ch 处的字节为 4Ah，指令：

```
ADD  A, [HL]
```

把累加器中的值（35h）与寄存器对 HL 寻址（107Ch）存储器得到的数值（4Ah）相加，并把计算结果（7Fh）保存到累加器。

在 8080 中，使用 ADC 指令和 SBB 指令可以对 16 位数、24 位数、32 位数甚至更高位的数进行加法、减法运算。例如，假设现在寄存器对 BC 和 DE 各自保存了一个 16 位的数，我们要把这两个数相加，并且把结果保存在寄存器对 BC 中。具体做法如下：

```
MOV A, C      ; 低字节操作
ADD A, E
MOV C, A
MOV A, B      ; 高字节操作
ADC A, D
MOV B, A
```

在上面的计算中，用 ADD 指令对低字节相加，用 ADC 指令对高字节相加。低字节相加产生的进位会进入高字节的运算中。在这段简短的代码中，我们用到了 4 个 MOV 指令，这是因为在 8080 中只能利用累加器进行加法运算，操作数在累加器和寄存器之间来回地传送，因此在 8080 的代码中会大量使用 MOV 指令。

现在我们来讨论 8080 的标志位（flag）。第 17 章设计的处理器已经有了 CF（进位标志位）和 ZF（零标志位）两个标志位，在 8080 中又新增了 3 个标志位，包括符号标志位 SF，奇偶标志位 PF 和辅助进位标志位 AF。在 8080 中，用一个专门的 8 位寄存器来存放所有标志位，该寄存器称做程序状态字（Program Status Word，PSW）。不同的指令对标志位有不同的影响，LDA、STA 或 MOV 指令始终都不会影响标志位，而 ADD、SUB、ADC 以及 SBB 指令会影响标志位的状态，具体情况如下。

- 如果运算结果的最高位是 1，那么符号标志位 SF 标志位置 1，表示该计算结果是负数。
- 如果运算结果为 0，则零标志位 ZF 置 0。
- 如果运算结果中“1”的位数是偶数，即具有偶数性（even parity），则奇偶标志位 PF 置 1；反之，如果“1”的位数是奇数，即运算结果具有奇数性（odd parity），则 PF 置 0。由于 PF 的这个特点，有时会被用来进行简单的错误检查。PF 在 8080 程序中并不常用。
- 进位标志位 CF 的情况和第 17 章描述的稍有不同，当 ADD 和 ADC 运算产生进位或者 SUB 和 SBB 运算不发生借位时，CF 都置 1。
- 辅助进位标志位 AF 只有在运算结果的低 4 位向高 4 位有进位时才置 1。它只用于 DAA（Decimal Adjust Accumulator，十进制调整累加器）指令中。

下面的两条指令会直接影响进位标志位 CF。

操作码	指　令	含　义
37	STC	令 CF 置 1
3F	CMC	令 CF 取反

第 17 章设计的计算机可以执行 ADD、ADC、SUB 和 SBB 指令（虽然缺乏灵活性），而 8080 功能更为强大，它还可以执行 AND（与）、OR（或）、XOR（异或）等逻辑运算。不论是算术运算还是逻辑运算，都是由 8080 处理器的算术逻辑单元（ALU）来完成的。

以下是 8080 的算术运算和逻辑运算指令。

操作码	指　令	操作码	指　令
A0	AND　A, B	B0	OR　A, B
A1	AND　A, C	B1	OR　A, C
A2	AND　A, D	B2	OR　A, D
A3	AND　A, E	B3	OR　A, E
A4	AND　A, H	B4	OR　A, H
A5	AND　A, L	B5	OR　A, L
A6	AND　A, [HL]	B6	OR　A, [HL]
A7	AND　A, A	B7	OR　A, A
A8	XOR　A, B	B8	CMP　A, B
A9	XOR　A, C	B9	CMP　A, C

AA	XOR A, D	BA	CMP A, D
AB	XOR A, E	BB	CMP A, E
AC	XOR A, H	BC	CMP A, H
AD	XOR A, L	BD	CMP A, L
AE	XOR A, [HL]	BE	CMP A, [HL]
AF	XOR A, A	BF	CMP A, A

AND、XOR 和 OR 都是按位运算（bitwise operations）指令，也就是说对于这些逻辑运算指令，其操作数的每一个对应位都是独立运算的，例如：

```
MVI   A, 0Fh
MVI   B, 55h
AND   A, B
```

保存到累加器的结果将会是 05h。假如我们把 3 条指令换作 OR，则最终的结果将会是 5Fh；如果换作 XOR，则结果又变成了 5Ah。

CMP（Compare，比较）指令同 SUB 指令类似，也是把两个数相减，不同之处在于它并不在累加器中保存计算结果，计算的目的是为了设置标志位。这个标志位的值可以告诉我们两个操作数之间的大小关系。例如，我们考虑下面的指令：

```
MVI   B, 25h
CMP   A, B
```

指令执行后，累加器 A 中的值并没有变化。改变的是标志位的值，如果 A 中的值等于 25h，则零标志位 ZF 置 1；如果 A 中的值小于 25h，则进位标志位 CF 置 1。

同样的，也可以对立即数进行这 8 种算术逻辑操作。

操作码	指　令	操作码	指　令
C6	ADI A, xx	E6	ANI A, xx
CE	ACI A, xx	EE	XRI A, xx
D6	SUI A, xx	F6	ORI A, xx
DE	SBI A, xx	FE	CPI A, xx

例如，可以用下面的这条指令来替代上面列出的两条指令：

```
CPI     A, 25h
```

下面是两种特别的 8080 指令。

操作码	指　令
27	DAA
2F	CMA

CMA 是 Complement Accumulator 的简写。它对累加器中的数按位取反，即把 0 变为 1，1 变为 0。例如，累加器中的数如果是 01100101，使用 CMA 命令后，累加器中的数按位取反，得到 10011010。我们还可以使用如下指令对累加器中的数取反：

```
XRI  A, FFh
```

前面提到过，DAA 是 Decimal Adjust Accumulator 的缩写，即十进制调整累加器，它可能是 8080 中最复杂的一条指令。在 8080 微处理器中专门设计了一个完整的小部件用来执行该指令。

DAA 指令提供了一种用二进制码表示十进制数的方法，称为 BCD 码（binary-coded decimal），程序员可以在该指令的帮助下实现十进制数的算术运算。BCD 码采用的表示方式为，每 4 位为一段，每段所能表示数的范围是：0000～1001，对应十进制数的 0～9。因为 1 字节有 8 位故可分割为 2 个段，因此在 BCD 码格式下，一个字节可以表示两位十进制数。

假设累加器 A 存放的是 BCD 码表示的 27h，显然它就对应十进制数的 27（通常，十六进制的 27h 对应的十进制数是 39）。同时假设寄存器 B 中存放着 BCD 码表示的 94h。假如执行如下指令：

```
MVI     A, 27h
MVI     B, 94h
ADD     A, B
```

累加器中存放的最终结果是 BBh，当然，这肯定不是 BCD 码。因为 BCD 码中每 4 位组成的段所能表示的十进制数不会超过 9。然而，当我们执行指令：

```
DAA
```

那么累加器最后所保存的值是 21h，而且进位标志位 CF 置 1。因为十进制的 27 与 94 相加的结果为 121。由此可以看到，使用 BCD 码进行十进制的算术运算是很方便的。

在 8080 程序中，经常会对一个数进行加 1 或减 1 运算。在第 17 章的乘法程序中，为了实现对一个数减 1，我们把该数与 FFh 相加，它是-1 的补码。8080 提供了专门的指

令用来对寄存器或存储器中的数进行加 1（称作增量）或减 1（称作减量）操作。

操作码	指　令	操作码	指　令
04	INR　B	05	DCR　B
0C	INR　C	0D	DCR　C
14	INR　D	15	DCR　D
1C	INR　E	1D	DCR　E
24	INR　H	25	DCR　H
2C	INR　L	2D	DCR　L
34	INR　[HL]	35	DCR　[HL]
3C	INR　A	3D	DCR　A

INR 和 DCR 都是单字节指令，它们可以影响除 CF（Carry Flag）之外的所有标志位。

8080 还包括 4 个循环移位（Rotate）指令，这些指令可以把累加器中的内容向左或向右移动 1 位，它们的具体功能如下。

操作码	指　令	意　义
07	RLC	使累加器循环左移
0F	RRC	使累加器循环右移
17	RAL	带进位的累加器循环左移
1F	RAR	带进位的累加器循环右移

这些指令只对进位标志位 CF 有影响。

假设累加器中存放的数是 A7h，即二进制的 10100111。RLC 指令使其每一位都向左移一位。最终的结果是，最低位（左端为低位，右端为高位）移出顶端移至尾部成为最高位，这条指令也会影响 CF 的状态。在这个例子中 CF 置 1，最后的结果为 01001111。RRC 指令以同样的方式进行右移位操作。如果移位之前的数是 10100111，执行 RRC 之后将变为 11010011，同时 CF 置 1。

较之 RLC 和 RRC，RAL 和 RAR 指令的工作方式稍有不同。执行 RAL 指令时，累加器中的数仍然按位左移，把 CF 中原来的值移至累加器中数值的最后一位，同时把累加器中数据的原最高位移至 CF。例如，假设累加器中移位之前的数是 10100111 且 CF 为 0，执行 RAL 指令后，累加器中的数变为 01001110 而 CF 变为 1。类似的，如果执行的是 RAR 指令，累加器中的数变为 01010011 而 CF 变为 1。

当我们在程序中需要对某个数进行乘 2（左移）或除 2（右移）运算时，使用移位操作会使运算变得非常简单。

我们通常把微处理器可以寻址访问的存储器称为随机访问存储器（random access memory，RAM），主要的原因是：只要提供了存储器地址（有多种方式），微处理器可以用非常简便的方式访问存储器的任意存储单元。RAM 就像一本书，我们可以翻到它的任意一页。这种方式很方便，它不像存储在一个微缩胶片上的整个星期的报纸，为了阅读星期六的内容，我们需要扫描几乎整个星期的内容。同样，它也比读取磁带快得多，因为当我们要听最后一首歌时，需要快进磁带的一整面。微缩胶片和磁带都不是随机访问的，它们是顺序访问（sequential access）的。

显然，随机访问存储器是非常好的一种寻址方式，对于经常访问存储器的微处理器来说更是如此。然而，在某些情况下使用不同的寻址方式访问存储器也是有好处的。例如，下面例子中的这种存储方式既不是随机的也不是顺序的：假设你在办公室工作，有人会到你办公桌前为你分配任务，每一项工作都用到某种文件夹。这些工作通常有这样的特点，在你完成某项工作之前首先要做另一项相关工作，并用到另一个文件夹。因此你只能放下第一个文件夹，并在它上面打开一个第二个文件夹继续工作。现在又有一个人给你分配了一个比前一项优先级更高的工作，于是你打开第三个文件夹放在前面两个上，继续工作。而这项工作也需要先做一项相关工作，于是你只好再打开第四个文件夹，现在你的办公桌上已经堆叠了四个文件夹了。

你可能已经注意到了，事实上，这些堆叠的文件夹很有序地保存了你干活的顺序轨迹。最上面的文件夹总是代表优先级最高的工作，完成该工作之后就可以做接下来的工作了，依此类推。最后当你处理完办公桌上最后一个文件夹（即接受的第一个任务）后，就可以回家了。

这种形式的存储器称作堆栈（stack）。使用堆栈时，我们以从底部到顶部的顺序把数据存入堆栈，并以相反的顺序把数据从堆栈中取出，因此该技术也称作后进先出存储器（last-in-first-out，LIFO）。堆栈的特点是，最先保存到堆栈中的数据最后被取出，而最后保存的数据则被最先取出。

同样，在计算机中也可以使用堆栈，当然计算机中的堆栈保存的是数据而不是工作。大量的实践证明，在计算机中使用堆栈技术是十分方便的。通常把将数据存入堆栈的过

程称作压入（push），把从堆栈取出数据的过程称作弹出（pop）。

假设你正在编写一个汇编语言程序，需要用到寄存器 A、B、C 来存储数据。在编写程序的过程中，你注意到程序需要做一个小的计算，并且该计算也需要用到寄存器 A、B、C。你希望在完成该计算之后仍然回到原来的地方，并且仍然使用寄存器 A、B、C 中原先存放的数据。

为了保存寄存器 A、B、C 原先存放的数据，可以简单地把这些数据保存到存储器中不同的地址中，需要进行的计算完成之后，再把这些数据从存储器转移到寄存器。但这种方式需要记录数据存放的地址。有了堆栈的概念之后，我们可以用一种更清晰的方式来处理这个问题，即把这些寄存器中的数据依次存放到堆栈中。

```
PUSH    A
PUSH    B
PUSH    C
```

稍后将具体解释这些指令实际的意义。现在需要知道的是，这些指令以某种方式把寄存器中的内容保存到先进后出存储器。这些指令执行之后，寄存器中原有的数据将妥善地保存下来，你就可以放心地使用这些寄存器进行别的工作了。为了取回原来的数据，可以使用 POP 指令把它们从堆栈中弹出，当然弹出的顺序和原来压入的顺序是相反的。相应的 POP 指令如下所示。

```
POP     C
POP     B
POP     A
```

再次谨记：后进先出。如果 POP 指令的顺序弄错了，将会引起严重的错误。

我们可以在程序中多次用到堆栈而不会引起混乱，这是堆栈机制的一个特殊优势。例如，在我们要编写的程序中已经把寄存器 A、B、C 中的数保存到了堆栈，在程序的另一段又需要把寄存器 C、D、E 中的数保存到堆栈，可以使用下面的指令：

```
PUSH    C
PUSH    D
PUSH    E
```

当然，在该段程序中还需要使用一些指令将保存到堆栈中的数据取回至寄存器，这些指令是：

```
POP     E
POP     D
POP     C
```

显然，由于先进后出的原则，这些数据在先前存放的 C、B、A 中的数据之前弹出堆栈。

堆栈的功能是怎样实现的呢？首先，堆栈其实就是一段普通的 RAM 存储空间，只是这段空间相对独立不另作他用。8080 微处理器设置了一个专门的 16 位寄存器对这段存储空间寻址，这个特殊的寄存器称为堆栈指针（SP，Stack Pointer）。

在我们所举的例子中，对于 8080 来说使用 PUSH 和 POP 对寄存器操作实际上是不准确的。在 8080 中，执行 PUSH 指令实际上是把 16 位的数据保存到堆栈，执行 POP 指令是把这些数据从堆栈中取回至寄存器。因此，对于上面的如 PUSH C，POP C 等指令，我们对其进行如下的修改。

操作码	指　令	操作码	指　令
C5	PUSH　BC	C1	POP　BC
D5	PUSH　DE	D1	POP　DE
E5	PUSH　HL	E1	POP　HL
F5	PUSH　PSW	F1	POP　PSW

PUSH　BC 指令将寄存器 B 和 C 中的数据保存到堆栈，而 POP　BC 则将这些数据从堆栈取回到寄存器 B 和 C 中，并且保持原来的顺序。最后一行指令中的 PSW 代表程序状态字，如前所述，这是一个 8 位的寄存器，用于保存标志位。最后一行的 PUSH 和 POP 指令的操作对象实际上是累加器和 PSW，即压入和弹出堆栈的数据由累加器和 PSW 中的内容组成。如果你想把所有寄存器中的数据及全部标志位都保存到堆栈，可以使用下面的指令：

```
PUSH    PSW
PUSH    BC
PUSH    DE
PUSH    HL
```

堆栈是怎样工作的呢？我们假设堆栈指针是 8000h，当执行 PUSH BC 指令时将会引发以下操作。

- 堆栈指针减 1，变为 7FFFh。

- 寄存器 B 中的内容被保存到堆栈指针指向的地址，即存储器地址 7FFFh 处。
- 堆栈指针减 1，变为 7FFEh。
- 寄存器 C 中的内容被保存到堆栈指针指向的地址，即存储器地址 7FFEh 处。

类似的，在堆栈指针仍为 7FFEh 的情况下，执行 POP BC 指令时会将上面的步骤反过来执行一遍：

- 堆栈指针指向的地址（7FFEh）的内容加载到累加器 C。
- 堆栈指针加 1，变为 7FFFh。
- 堆栈指针指向的地址（7FFFh）的内容加载到累加器 B。
- 堆栈指针加 1，变为 8000h。

每执行一条 PUSH 指令，堆栈都会增加两个字节，这可能会导致程序出现一些小错误——堆栈可能会不断增大，最终覆盖掉存储器中保存的程序所必需的代码或数据。这种错误被称作堆栈上溢（stack overflow）。类似的，如果在程序中过多地使用了 POP 指令，则会过早地取完堆栈中的数据从而导致类似的错误，这种情况称为堆栈下溢（stack underflow）。

如果 8080 连接的是一个 64 KB 的存储器，你可能会把堆栈指针初始化为 0000h。当执行第一条 PUSH 指令时，堆栈指针会减 1 变为 FFFFh，即存储器的最后一个的存储单元。这时，堆栈的初始位置将会是存储器的最高地址，因为程序的代码通常从 0000h 开始存放，因此两者将保持非常远的距离。

8080 使用 LXI 指令为堆栈寄存器赋值，LXI 是 Load Extended Immediate 的缩写，即加载扩展的立即数。下面的这些指令将把操作码后的两个字节保存到 16 位寄存器对中。

操作码	指　令
01	LXI　BC, xxxx
11	LXI　DE, xxxx
21	LXI　HL, xxxx
31	LXI　SP, xxxx

指令：

```
LXI  BC, 527Ah
```

和下面两条指令等价：

```
MVI  B, 52h
MVI  C, 7Ah
```

LXI 指令保存一个字节。而且上表中最后一条 LXI 指令为堆栈指针赋了一个特殊的值。通常下面的指令不作为微处理器复位之后首先执行的指令之一。

```
0000h:        LXI  SP, 0000h
```

类似的，还可以对寄存器对和堆栈指针进行加 1 或减 1 操作，即把它们看做 16 位寄存器。

操作码	指　令	操作码	指　令
03	INX　BC	0B	DCX　BC
13	INX　DE	1B	DCX　DE
23	INX　HL	2B	DCX　HL
33	INX　SP	3B	DCX　SP

对于要讨论的 16 位指令，我们可以再看一些例子。下面的指令把由任意 2 个寄存器组成的 16 位寄存器对的内容加到寄存器对 HL 中。

操作码	指　令
09	DAD　HL, BC
19	DAD　HL, DE
29	DAD　HL, HL
39	DAD　HL, SP

这些指令可以减少操作的字节数。例如，上面的第一条指令一般情况下需要 6 个字节。

```
MOV  A, L
ADD  A, C
MOV  L, A
MOV  A, H
ADC  A, B
MOV  H, A
```

DAD 指令一般用来计算存储器地址，只对进位标志位 CF 有影响。

接下来我们来认识一下各种各样的指令。下面两条指令的特点是操作码后面都跟着 2 字节的地址，第一条指令把 HL 寄存器对的内容保存到该地址，第二条指令把该地址的内容加载到 HL 寄存器对。

操作码	指　令	意　义
22h	SHLD　[aaaa], HL	直接保存 HL 中的数据
2Ah	LHLD　HL, [aaaa]	直接加载数据到 HL

寄存器 L 的数据保存在地址 aaaa，而寄存器 H 的数据保存在地址 aaaa+1。

下面的两条指令把寄存器对 HL 保存的数据加载到程序计数器和堆栈指针。

操作码	指　令	意　义
E9h	PCHL　PC, HL	将 HL 保存的数据加载到程序计数器
F9h	SPHL　SP, HL	将 HL 保存的数据加载到堆栈指针

PCHL 指令本质上是一种 Jump 指令，它把 HL 保存的存储器地址加载到程序计数器，而 8080 处理器要执行的下一条指令就是程序计数器所指明的存储器地址中存放的指令。SPHL 指令可以作为另一种为堆栈指针赋值的指令。

下面的两条指令中，第一条将 HL 保存的数据与堆栈顶部的两个字节进行交换；第二条指令将 HL 保存的数据和寄存器对 DE 保存的数据进行交换。

操作码	指　令	意　义
E3h	XTHL　HL, [SP]	把 HL 中的内容和堆栈顶部 2 个字节进行交换
EBh	XCHG　HL, DE	把 DE 中的内容和 HL 中的内容进行交换

除了刚讲过的 PCHL 指令外，目前为止还没有介绍过 8080 的跳转指令。如第 17 章所述，在处理器中专门设置了一个称为程序计数器 PC 的寄存器，它用来保存处理器将要取出并执行的指令的存储地址。通常，处理器在 PC 的指引下顺序执行存储器中存放的指令，但有一些指令，如 Jump（跳转）、Branch（分支）或 Goto（无条件转移）——使处理器脱离原来的执行顺序。这些指令使 PC 重新加载另外的值，处理器要执行的下一条指令存放于存储器的其他位置，而不在按原来的顺序寻址。

当然，原始的普通跳转指令的确有一定的作用，但从第 17 章得来的经验可以知道，条件跳转（conditional jump）指令的作用更大。条件跳转指令使处理器根据某些标志位的

值转移到特定的地址，这些标志位可以是进位标志位 CF、零标志位 ZF 等。正是由于条件跳转指令的引入，第 17 章所设计的自动加法器才成为一般意义上的数字计算机。

8080 有 5 个标志位，其中有 4 个可用于条件跳转指令。8080 支持 9 种不同的跳转指令，包括了非条件跳转指令，还包含根据 ZF(Zero Flag)、CF(Carry Flag)、PF(Parity Flag) 以及 SF（Sign Flag）是否为 1 而跳转的条件跳转指令。

在介绍这些指令之前，首先来介绍与 Jump 指令相关的另外两种指令。第一种是 Call（调用）指令，它和 Jump 指令类似，但是有所不同的是：执行 Call 指令后，程序计数器（Program Counter，在这部分讲解中简称 PC）加载一个新的地址，而处理器会把原来的地址保存起来，保存到何处呢？最好的选择自然是堆栈了。

这种策略使 Call 指令有效地记录了“从何处跳转”(where it jumped from)，即保存了跳转之前的相关信息。堆栈中保存的地址可以使处理器最后返回到转移之前的位置。用于返回的指令称为 Return（返回）。Return 指令从堆栈中弹出两个字节，并把它们加载到 PC 中，这样就完成了返回到跳转点的工作。

对于任何处理器来说，Call 和 Return 指令都非常重要。在它们的帮助下，程序员可以在程序中使用子程序（subroutine），子程序是一段频繁使用的完成特定功能的代码（这里的“频繁”意味着“不止一次”）。对于汇编语言来说，子程序是其基本的组成部分。

让我们来看一个使用子程序的例子。假设你在编写一个汇编语言程序，在程序的某个位置你需要把两个字节相乘，因此你编写了一段用于两个数相乘的代码，然后继续向下写，在程序的另一个位置你发现需要再一次对两个字节相乘。因为你已经写过把两个字节相乘的代码，所以只需要重复使用这些代码就可以了。但是怎么做呢？只是简单地把这些代码重复输入到存储器吗？我们希望不是，因为这样做不仅耽误时间而且浪费存储空间，一个更好的方法是跳转到先前写的那段乘法代码所在的位置。但是普通的 Jump 指令不能完成这个操作，因为在执行乘法之后不能准确地返回程序的当前位置。因此你需要使用 Call 指令和 Return 指令来帮助你实现这个功能。

用来实现两个数相乘的一组指令可以作为一个子程序。下面我们将看到这个子程序。在第 17 章的乘法程序中，被乘数（还有乘积）被保存在存储器的特定位置；而在 8080 的子程序中，乘数和被乘数分别存放在寄存器 B 和寄存器 C 中，乘积保存到 16 位寄存器

对 HL 中。8080 中的乘法子程序如下：

```
Multiply:   PUSH    PSW         ; 将要修改的寄存器的原内容保存至堆栈
            PUSH    BC

            SUB     H, H        ; 将 HL（即乘积）置为 0000h
            SUB     L, L

            MOV     A, B        ; 乘数送至累加器 A
            CPI     A, 00h      ; 如果累加器中的值是 0，则结束
            JZ      AllDone

            MVI     B, 00h      ; 将 BC 的高字节置为 0

MultLoop:   DAD     HL, BC      ; 将 BC 的内容加到 HL
            DEC     A           ; 乘数减 1
            JNZ     MultLoop    ; 如果不为 0，则跳转

AllDone:    POP     BC          ; 将堆栈中保存的数据恢复至寄存器
            POP     PSW
            RET                 ; 返回
```

注意，上述子程序的第一行有一个标志 Multiply。当然，实际上这个标志对应着子程序在存储器中的起始地址。该子程序在开始处使用了两个 PUSH 指令，这是因为通常在子程序的起始处要保存程序用到的寄存器。

保存寄存器后，子程序下面要做的是把寄存器 H 和 L 置 0。尽管可以使用 MVI（转移立即数）指令代替 SUB 指令来实现该操作，但这样会用到 4 个字节而不是 2 个字节的指令。子程序执行成功后，运算结果会保存到寄存器对 HL 中。

接下来子程序把寄存器 B 中的数（即乘数）转移到累加器 A，并判断该数是否为 0。如果为 0，则乘法子程序结束，因为乘数为 0。由于寄存器 H 和 L 已经为 0，所以子程序可以使用 JZ（Jump If Zero）指令跳转到程序最后的两条 POP 指令。

如果乘数不是 0，子程序会把寄存器 B 置为 0。现在寄存器对 BC 存放的是 16 位的被乘数，而累加器中存放的是乘数。接下来 DAD 指令会把 BC（被乘数）加到 HL（运算结果）中。A 中的乘数减 1，如果结果不为 0，则执行 JNZ（非零跳转）指令，该指令会使 BC 再次加到 HL。这个循环会继续执行，直到循环的次数等于乘数为止（当然也可以

利用 8080 的移位指令编写一个更有效率的乘法子程序）。

在程序中使用如下指令来调用这个乘法子程序，例如，把 25h 和 12h 相乘：

```
MVI     B, 25h
MVI     C, 12h
CALL    Multiply
```

CALL 指令把 PC 的值保存到堆栈中，被保存的这个值是 CALL 指令的下一条指令的地址，然后 CALL 指令将使程序跳转到标志为 Multiply 的指令，即子程序的起始处。当子程序得到计算结果后，执行 RET（返回）指令，该指令使保存在堆栈的 PC 的值弹出，并重新设置到 PC，之后程序将继续执行 CALL 指令后面的指令。

8080 指令集包括条件 CALL 指令和条件 Return 指令，但它们使用的频率比条件跳转指令小得多。下面的表格完整地列出了这些指令。

条件	操作码	指令	操作码	指令	操作码	指令
None	C9	RET	C3	JMP aaaa	CD	CALL aaaa
Z not set	C0	RNZ	C2	JNZ aaaa	C4	CNZ aaaa
Z set	C8	RZ	CA	JZ aaaa	CC	CZ aaaa
C not set	D0	RNC	D2	JNC aaaa	D4	CNC aaaa
C set	D8	RC	DA	JC aaaa	DC	CC aaaa
Odd parity	E0	RPO	E2	JPO aaaa	E4	CPO aaaa
Even parity	E8	RPE	EA	JPE aaaa	EC	CPE aaaa
S not set	F0	RP	F2	JP aaaa	F4	CP aaaa
S set	F8	RM	FA	JM aaaa	FC	CM aaaa

正如你大概所了解的，存储器并不是连接在微处理器上的唯一设备。一个完整的计算机系统通常需要输入/输出设备（I/O）以实现人机交互。输入/输出设备通常包括键盘和显示器等。

微处理器是如何与外围设备（peripheral，除存储器外，与微处理器连接的所有设备都可以称为外围设备）互相通信的呢？外围设备配备了与存储器类似的接口，微处理器通过与某种外围设备对应的特定地址（即接口）对其进行读写操作。在某些微处理器中，外围设备实际上占用了一些通常用来寻址存储器的地址，这种结构称作内存映像 I/O（memory-mapped I/O）。但在 8080 中，除了常规的 65536 个地址外，另外增加了 256 个地

址专门用来访问输入/输出设备，它们被称作 I/O 端口（I/O ports）。I/O 地址信号标记为 $A_0 \sim A_7$，但 I/O 的访问方式与存储器的访问方式不同，两者的区分由 8228 系统控制芯片的锁存信号来标识。

OUT（输出）指令把累加器中的内容写入到紧跟该指令后的字节所寻址的端口（port）。IN（输入）指令把一个字节从端口读入到累加器。它们的格式如下所示。

操作码	指令
D3	OUT PP
DB	IN PP

外围设备有时候需要获得处理器的注意。例如，当你按下键盘的某个键时，处理器应该马上注意到这个事件。这个过程由一个称为中断（interrupt）的机制实现，这是一个由外围设备产生的信号，连接至 8080 的 INT 输入端。

但是，当 8080 复位后，就不再响应中断。程序必须执行 EI（Enable Interrupt）指令来允许中断，然后执行 DI（Disable Interrupts）禁止中断。这两条指令如下所示。

操作码	指令
F3	DI
FB	EI

8080 的 INTE 输出信号用来指明何时允许中断。当外围设备需要中断微处理器的当前工作时，它需要把 8080 的 INT 输入信号置为 1。8080 通过从存储器中取出指令来响应该中断，同时控制信号指明有中断发生。外围设备通常提供下列指令来响应 8080 微处理器。

操作码	指令	操作码	指令
C7	RST 0	E7	RST 4
CF	RST 1	EF	RST 5
D7	RST 2	F7	RST 6
DF	RST 3	EF	RST 7

上面列出的这些指令都称作 Restart（重新启动）指令，在其执行的过程中也会把当前 PC 中的数据保存到堆栈，这一点与 CALL 指令类似。但 Restart 指令在保存 PC 数据之后会立刻跳转到特定的地址，而且是根据参数的不同将跳转到不同的地址：比如 RST 0

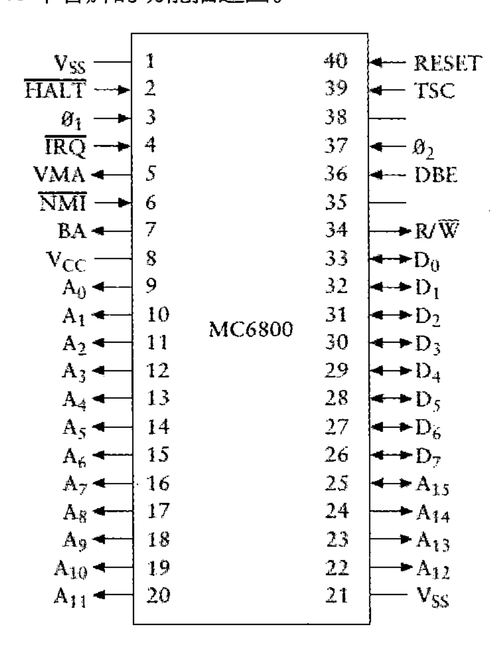
将跳转到地址 0000h 处，RST 1 将跳转到地址 00008h 处，依此类推，最后的 RST 7 将跳转到地址 0038h 处。这些地址存放的代码都是用来处理中断的。例如，由键盘引起的中断将执行 RST 4 指令，程序将跳转到地址 0020h 处，该地址存放的代码将负责从键盘读入数据（完整的过程将在第 21 章讲述）。

目前为止，我们已经介绍了 243 个操作码。在前 255 个数中，有 12 个没有作为操作码使用，它们是：08h，10h，18h，20h，28h，30h，38h，CBh，D9h，DDh，EDh 和 FDh。下面还需要讲到一个操作码。

操作码	指令
00	NOP

NOP 代表（即声明）no op（no operation，无操作）。NOP 指令使处理器什么操作也不执行。这样做有什么好处呢？填空，即保持处理器的运行状态而不做任何事情。8080 可以执行一批 NOP 指令而不会引起任何错误事件的发生。

本章不准备详细介绍 Motorola 6800 微处理器，因为在构造和功能方面它与 8080 非常相似。下面是 6800 的 40 个管脚的功能描述图。



在上图中，V_{SS} 表示接地，V_{CC} 代表 5V 的电源。同 8080 一样，6800 也有 16 个地址

输出信号端和 8 个数据信号端，其中数据信号端既可以用于输入信号也可以用于输出信号。它还有一个 RESET 信号端和一个 R/$\overline{\text{W}}$（read/write，读/写）信号。$\overline{\text{IRQ}}$ 信号代表中断请求。与 8080 相比，6800 的时钟信号较为简单，6800 没有设计独立的 I/O 端口，所有的输入/输出设备的地址都是存储器地址空间的一部分。

6800 有一个 16 位的程序计数器 PC、一个 16 位的堆栈指针 SP、一个 8 位的状态寄存器（用来保存标志位），以及两个 8 位的累加器 A、B。A 和 B 都可以用做累加器（而不是把 B 作为普通的寄存器），因为 A 和 B 的功能完全相同，任何用 A 做的工作都可以用 B 实现。与 8080 不同，6800 没有设置其他的 8 位寄存器。

6800 设置了一个 16 位的索引寄存器（index register），它可以用来保存 16 位的地址，其功能与 8080 的 HL 寄存器对相似。对于 6800 的大部分指令来说，它们的地址都可以由索引寄存器与紧跟在操作码后的字节相加得到。

尽管 6800 实现的操作与 8080 大致相同——加载、保存、加法、减法、移位、跳转、调用等，但对应的操作码和助记符是完全不同的。例如，下面列出了 6800 的转移（Branch）指令集。

操作码	指　令	意　义
20h	BRA	转移
22h	BHI	大于则转移
23h	BLS	相等或小于则转移
24h	BCC	进位为 0 则转移
25h	BCS	进位为 1 则转移
26h	BNE	不相等则转移
27h	BEQ	相等则转移
28h	BVC	溢出置 0 则转移
29h	BVS	溢出置 1 则转移
2Ah	BPL	为正数则转移
2Bh	BMI	为负数则转移
2Ch	BGE	大于或等于 0 则转移
2Dh	BLT	小于 0 则转移
2Eh	BGT	大于 0 则转移
2Fh	BLE	小于或等于 0 则转移

与 8080 不同，6800 没有设置奇偶标志位，而是设置了一个溢出标志位（Overflow flag）。上面的转移指令中有一些依赖于标志位的组合（combinations of flags）。

当然，8080 和 6800 的指令集是不同的，虽然这两款芯片于同一年发布，但它们是由属于不同公司的两组不同的工程师设计的。这就造成了它们之间的不兼容，因此它们不能执行对方的机器码，为一种芯片编写的汇编语言程序也不能在另一种芯片上执行。如何编写能在不同类型处理器上执行的计算机程序是第 24 章的主题。

8080 和 6800 的另一个有趣的区别是：在两个处理器中，LDA 指令都从存储器的特定地址将数据加载到累加器。例如，在 8080 中，下面的字节序列：

3Ah	8080 LDA 指令
7Bh	
34h	

将把存储器地址 347Bh 处的字节加载到累加器。现在对比一下 6800 的 LDA 指令，它使用 6800 扩展寻址模式（6800 extended addressing mode）：

B6h	6800 LDA 指令
7Bh	
34h	

上面的这组字节序列将把存储器 7B34h 地址处的字节加载到累加器。

两者的区别是很微妙的。当然，你可能已经注意到了操作码的不同：8080 的操作码是 3Ah，而 6800 的操作码是 B6h。但是两种微处理器对紧跟在操作码后的地址的处理方式是不同的，8080 假设低字节在前，高字节在后；而 6800 假设高字节在前，低字节在后。

Intel 和 Motorola 的微处理器在保存多字节数据问题上的根本区别从未得到解决。直到今天，英特尔的微处理器在保存多字节数据时，仍然把最低有效字节放在最前面（也就是说，在最低地址处），而 Motorola 的微处理器在保存多字节数据时，仍然把最高有效字节放在最前面。

这两种不同的方式分别称为 little-endian（Intel 方式）和 big-endian（Motorola 方式）。

争论两者之间哪一种方式更好是件有趣的事，但在这么做之前，先要知道 big-endian 这个术语出自乔纳森·斯威夫特（Jonathan Swift）的 *Gulliver's Travels*，指的是刘普特（Lilliput）和布鲁夫思科（Belfuscu）之间关于在吃鸡蛋之前应该把鸡蛋的哪一头敲碎的争论。因此，这种争论可能是没有意义的（另一方面，坦白地说，在本书第 17 章设计的计算机所采用的方式我个人并不喜欢）。尽管不能确定那一种方式本质上是“对的”，这种差别确实造成了附加的兼容性问题，这种问题通常会在采用 little-endian 和 big-endian 系统的机器共享信息时出现。

这两种微处理器后来的发展如何呢？8080 被应用在一些人所谓的第一台个人电脑（personal computer）上，更准确地说应该是用于第一台家用电脑（home computer）上。下图是 Altair 8800，它曾登上了 1975 年 1 月的 *Popular Electronics* 杂志的封面。



当你看到 Altair 8800 时，前面板上的灯泡和开关会让你感到似曾相识。这个界面和第 16 章介绍 64 KB RAM 阵列时的初始“控制面板”的界面是类似的。

在 8080 之后，Intel 又推出了 8085 芯片，而具有更重大意义的是 Z-80 芯片的出现。Z-80 是由 Zilog 公司制造的，该公司是英特尔公司的竞争对手，由英特尔的前雇员费德瑞克·菲戈金（Federico Faggin）创立，费德瑞克·菲戈金曾在 4004 芯片的研制过程中做出重要贡献。Z-80 和 8080 完全兼容，并且增加了许多非常有用的指令。1977 年，Z-80 曾被应用在 Radio Shack TRS-80 Model 1 上。

同样是在 1977 年，由斯蒂夫·乔布斯（Steven Jobs）和史蒂芬·沃兹内卡（Stephen Wozniak）创立的苹果计算机公司推出了新一代产品 Apple II。Apple II 既没有使用 8080

也没有使用 6800，而是使用了基于 MOS 技术的更加便宜的 6502 芯片，它是 6800 的改进加强版本。

1978 年 6 月，英特尔公司推出了 8086 芯片，这是一个 16 位的微处理器，可以寻址 1MB 的地址空间。8086 的操作码与 8080 不兼容，但它包含了乘法指令和除法指令。一年后，英特尔推出了 8088 芯片，其内部结构与 8086 完全相同，但在外部仍以字节为单位（即外部接口为 8 位）访问存储器，所以该芯片能使用为 8080 设计的较为流行的 8 位的外围芯片（8-bit support chips）。IBM 在 5150 个人计算机中使用了 8088 芯片，这种计算机通常称为 IBM PC，于 1981 年秋季推出。

IBM 大举进军个人计算机（Personal Computer，有时也简称为 PC）市场对业界产生了重大影响，许多公司都推出了与个人计算机兼容的机器（兼容的含义将在随后的几章里详细讨论）。多年以来，"IBM PC 兼容"也暗示了"Intel inside"（即内部使用了 Intel 微处理器），这里特指 Intel x86 系列微处理器。x86 系列微处理器包括 1982 年发布的 186 芯片和 286 芯片，1985 年发布的 32 位 386 芯片，1989 年发布的 486 芯片。从 1993 年开始，英特尔公司推出 Intel 奔腾（Intel Pentium）系列微处理器，而这个系列如今被广泛地应用于 PC 兼容机。虽然这些处理器的指令集都在不断扩展，但是它们仍然支持始于 8086 的所有早期处理器的操作码。

苹果公司的 Macintosh 于 1984 年首次发布，它采用摩托罗拉的 68000 微处理器，68000 是 16 位微处理器，是 6800 的下一代产品。68000 及其后续产品（通常称为 68K 系列）是已发布的处理器中最受欢迎的一类。

从 1994 年开始，Macintosh 计算机开始使用 PowerPC 微处理器，该处理器是由摩托罗拉，IBM 以及苹果公司联合开发的。PowerPC 是采用 RISC（Reduced Instruction Set Computing，精简指令集计算机）微处理器体系结构来设计的，其目的是通过某些方面的简化来提高处理器的速度。在 RISC 计算机中，通常指令都是等长的（PowerPC 中是 32 位），只有加载和保存两种指令能访问存储器，并且尽量简化指令的操作。RISC 处理器设置了大量的寄存器，这样就能避免频繁访问存储器以提其高运行速度。

PowerPC 拥有完全不同的指令集，因此不能执行 68K 系列微处理器的代码。然而，目前 Macintosh 计算机使用的 PowerPC 微处理器可以仿真（emulate）68K 系列微处理器。运行在 PowerPC 上的仿真程序逐一检查 68K 程序的操作码，并执行相应的操作。它执行

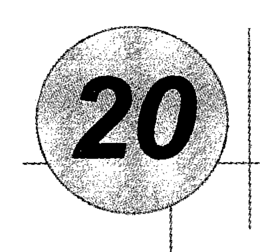
的速度没有 PowerPC 本身的代码那么快，但可以正常工作。

根据摩尔定律（Moore's Law），微处理器中的晶体管数量每 18 个月翻一倍，人们不禁要问：增加的这些大量的晶体管用来做什么呢？

一些晶体管用来适应处理器不断增加的数据宽度——从 4 位、8 位、16 位到 32 位；另一些新增的晶体管用来应对新的指令。例如，现在大部分微处理器都支持用于浮点数的指令（将在第 23 章详细介绍）；还有一些新增的指令用来执行重复计算，以便在计算机屏幕上呈现图片和电影。

现代处理器使用多种技术来提高其运行速度。其中一种就是流水线技术（pipelining），即处理器在执行一条指令的同时读取下一条指令，尽管 Jump 指令在一定程度上会改变这种流程。现代处理器还包括一个 Cache（高速缓冲存储器），它是一个设置在处理器内部，访问速度非常快的 RAM 阵列，用来存放处理器最近要执行的指令。由于计算机程序经常执行一些小的指令循环，使用 Cache 可以避免反复加载这些指令。上面提到的这些提高运行速度的策略都需要在处理器内部增加更多的逻辑组件和晶体管。

正如前面所提到的，微处理器只是整个计算机系统的一部分（尽管是最重要的一部分）。我们会在第 21 章构造这样一个系统，但首先要学习如何处理存储器中的数据，包括操作码和数字，我们要对这些数据进行编码。让我们从心态上回归到小学一年级，像孩子们学习读写一样学习如何编码吧。



ASCII码和字符转换

数字计算机中的存储器唯一可以存储的是比特，因此如果要想在计算机上处理信息，就必须把它们按位存储。通过先前的学习，我们已经掌握了如何用比特来表示数字和机器码。现在我们面临的一大挑战就是如何用它来存储文本。毕竟，人类所积累的大部分信息，都是以各种文本形式保存的。文本信息聚集最多的地方之一就是图书馆，数不清的书、杂志和报纸所提供的都是文本信息。当然，我们现在已经使用计算机来存放图像和影音信息了，不过为了易于理解，我们还是先从如何使用计算机存放文本开始讲解。

为了将文本表示为数字形式，我们需要构建一种系统来为每一个字母赋予一个唯一的编码。数字和标点符号也算做文本的一种形式，所以它们也必须拥有自己的编码。简而言之，所有由符号所表示的字母和数字（Alphanumeric）都需要编码。具有这种功能的系统被称为字符编码集（Coded Character Set），系统内的每个独立编码称为字符编码（Character Codes）。

许多疑问也随之而来，而要解决的第一个问题是：构成这些编码究竟需要多少比特？

要想回答这个问题就需要我们从长计议了。

当考虑用比特来表示文本的时候，切忌好高骛远。我们经常会看到书页上，或报刊和杂志的栏目上所有的内容被整齐地组织在一起。所有的段落都划分为宽度相等的文本行，但我们要注意，这种排版的形式永远只是文本之外的事物。当曾在某个杂志上细细品味过的一个故事，几年后与我们在另外一本书中重逢时，我们回忆起的往往是故事本身而不是文本的排版，没有人会因为行与行间距的不同而把它们当成两个故事。

我极力想阐述一个重要的观点，那就是文本与其印刷在纸上时采用的二维排版格式是两码事。充分发挥想象力，将文本看成是一维的由字母、数字和标点符号组成的数据流吧。当然，有时为了标明一句话的开始和结尾，还需要一些额外的编码。

还是先前所描述的例子，曾在某个杂志上细细品味过的一个故事，几年后出现在另外一本书中，但是文章的字体发生了变化，这算是一个问题吗？当年的杂志上是这样印刷的：

Call me Ishmael.

而现在书中的写法变成了下面这样：

Call me Ishmael.

这些区别是我们所在意的吗？答案往往是否定的。字体改变了文本的表现形式，但故事本身的内容并没有因此而改变。字体是可以变来变去的。但这并无大碍。

还有一种简化问题的方法：我们可以总是使用毫无修饰的文本。没有斜体、粗体、下画线、颜色、空心体、上标、下标以及音标，同样的，这里没有元音字母标识等符号，只有赤条条的拉丁字母，这些字母组成了英语中 99%的文本。

在先前对莫尔斯码和布莱叶盲文的学习中，我们了解了如何将字母表中的字符以二进制的形式表现出来。这些系统在适合的场合很好用，但要想用到计算机中却是难上加难。就拿莫尔斯码来说，它是变量自适应长度（Variable-Width）编码：常用字符的编码较短，而不常用字符的编码较长。这样的编码非常适合电报系统，但并不适用于计算机。另外，莫尔斯码并不区分字母的大小写。

布莱叶盲文编码使用固定宽度，非常适合计算机使用。每一个字符对应着 6 比特的

编码，并且用到了转义（Escape）码对大小写进行了区分。转义码用来表明下一个字符为大写。这也就是说，每个大写字母都需要两组编码来表示。布莱叶盲文中用移位（Shift）码表示数字：移位码后紧跟的编码都被看做数字，直到遇到下一个移位码，此时系统又将后面的内容当做字母。

我们的目标是开发一个字符编码集，使用这个编码集，系统可以将如下的句子转换成为一系列的编码：

I have 27 sisters.

每一个字符的编码都会占据一定的比特。有的编码用来表示字母，有的用来表示标点符号，还有一些用来表示数字。甚至于单词间的空格也需要单独的编码。上面的句子中共 18 个字符（包括字间空格），对这样一个句子进行编码后得到的连续字符通常被称为文本字符串（string）。

我们需要对字符串中的数字进行编码，例如上面的句子中的 27。或许大家会感到疑惑，因为之前我们都是用比特来表示数字的。最简单的，也是最容易想到的做法就是使用二进制数 10 和 111 作为 2 和 7 的编码。但是这里却不适用。在这个句子中，可以像处理其他的字符一样来处理 2 和 7。它们的编码可以和本身表示的含义无关。

1874 年由法国电报服务公司（French Telegraph Service）职员埃米尔·波多（Emile Baudot）发明了可以打印的电报机，划时代的波多电传码也应运而生。即使在今天来看，这种编码十分“经济划算”，每一个文本字符都采用 5 位编码。这种编码 1877 年被法国电报服务公司采纳，后来经唐纳德·默里（Donald Murray）修改，最终在 1931 年被当年的 CCITT 组织（Comité Consultatif International Télégraphique et Téléphonique），即现在的国际电信联盟（ITU）定为标准。该编码的正式名称是国际电报字母表第二号（International Telegraph Alphabet No.2）或 ITA - 2，在美国常常被称为波多印字电报制（Baudot），不过更准确地说，叫做默里（Murray）编码。

随着 20 世纪的到来，Baudot 被广泛应用于电传打字机（teletypewriters）。Baudot 电传打字机配备了一个输入键盘，这款键盘有些像打字机，但只有 30 个键和一个空格键。电传打字机键盘上的每一个键实质上都起到了转换器的作用，它负责产生二进制编码并且通过输出电缆逐位传输出去。电传打字机也具备打印功能，通过输入电缆读取编码，

触发电磁铁，从而将字符打印在纸上。

由于 Baudot 对每个字符采用 5 位编码，整个系统由 32 个编码所组成，这些编码的十六进制取值范围从 00h 到 1Fh。下表给出了 32 个不同编码的十六进制形式及其所对应的字母表中的字符。

十六进制码	Baudot 字符	十六进制码	Baudot 字符
00		10	E
01	T	11	Z
02	回车	12	D
03	O	13	B
04	空格	14	S
05	H	15	Y
06	N	16	F
07	M	17	X
08	换行	18	A
09	L	19	W
0A	R	1A	J
0B	G	1B	数字转义符号
0C	I	1C	U
0D	P	1D	Q
0E	C	1E	K
0F	V	1F	字符转义符号

编码 00h 被保留了下来，没有指派给任何值。剩下的 31 个编码中，字母表中的字符占了 26 个，其余 5 个用来调整格式，如上表中的楷体排版的语句所示。

编码 04h 用来表示空格，通常用于分隔单词。编码 02h 和 08h 表示的是回车和换行。这些都是电传打字机中的专用术语。当使用电传打字机上打字，一旦到了一行的末尾时，我们通常会按下一个操作杆或按钮。这个操作其实包括两个动作：第一个动作是，使打印机的滑架回到起始位置，这样打印下一行时可以从纸的最左边开始，这就是回车。第二个动作是，将打印机的滑架移至正在使用中的位置的下一行，这就是换行。在 Baudot 编码系统中，这两个编码由专门的按键产生。Baudot 电传打字机在打印的时候会响应这两个编码以完成相应的操作。

Baudot 系统里怎么没有数字和标点符号呢？其实这是因为编码 1Bh 中暗藏玄机，它

的实际作用是数字转义（Figure Shift）。数字转义编码后的所有的编码都会被解释为数字或标点符号，直到遇到字符转义编码（1Fh），一切就又被解释为字符。下表展示了十六进制编码以及所对应的数字和标点符号。

十六进制码	Baudot 字符	十六进制码	Baudot 字符
00		10	3
01	5	11	+
02	回车	12	身份不明
03	9	13	?
04	空格	14	'
05	#	15	6
06	,	16	$
07	.	17	/
08	换行	18	-
09	)	19	2
0A	4	1A	响铃
0B	&	1B	数字转义符号
0C	8	1C	7
0D	0	1D	1
0E	:	1E	(
0F	=	1F	字符转义符号

其实在 ITU 规范化的编码方案中，05h、0Bh 和 16h 是留做他用的，官方说法为“国内使用”。表中列出的是这几个编码在美国使用时的含义。某些欧洲国家将这些编码代表重音符号。响铃编码令电传打字机发出清脆的铃声。“Who Are You”编码用来让打字员激活身份识别机制。

像莫尔斯码一样，这种 5 位的编码并没有提供区分大、小写的方法。下面这个句子：

I SPENT $25 TODAY.

表示成编码的十六进制数据流就是：

0C 04 14 0D 10 06 01 04 1B 16 19 01 1F 04 01 03 12 18 15 1B 07 02 08

请注意三个转义码的使用：1Bh 出现在数字之前，1Fh 出现在数字之后，而数字结束之后又出现了 1Bh。这一行编码以回车、换行符结尾。

问题出来了，如果把相同的数据流再一次输入到电传打印机，情况就大不一样了，

如下所示：

I SPENNT $25 TODAY.

8'03,5 $25 TODAY.

怎么会这样？这是由于在接收到第二行编码之前打印机接收到的最后一个转义码是数字转义码，所以当遇见第二行开头几个编码时，打印机将它们解释成数字。

这种问题产生的根源就是采用了转义码，这的确很让人头痛。尽管 Baudot 电传码是很简洁实用的编码，但是，我们更加希望采用能唯一表示字符、数字及标点符号的编码方案，如果还能对大、小写进行区分那就更好不过了。

如果想知道比 Baudot 更好用的编码系统中一个编码需要多少比特，我们需要做几个小加法：所有的大小写字母加起来共需 52 个编码，0～9 数字需要 10 个编码，加起来共有 62 个，如果算上一些标点符号，数量超过了 64 个，也就是说，一个编码至少需要 6 比特。但无论如何字符数应该不超过 128 个，而且应该远远不够 128 个，也就是说编码长度不会超过 8 位。

所以，答案就是 7。在采用 7 位编码时，不需要转义字符，而且可以区分字母的大小写。

这些字符编码是什么样子的呢？其实我们可以随意编码。如果我们要去打造一台自己的计算机，计算机硬件的每一个部分都要亲自制作，计算机内部的程序也要亲手编写，而且不打算把这台计算机与其他的进行连接，那么就完全可以构造自己的编码系统。其实也很简单，就是给每一个字符指派一个唯一的编码。

但是这种自己制造计算机，并且独立使用的情况实在是太少了，所以所有人都遵循并使用统一化的编码，计算机的存在才有意义。这样一来，使用不同方法制造出的计算机之间就可以互相兼容，甚至可以互相交流文本信息。

这样一来，随意的编码就显得不太合适了。当我们使用计算机来处理文本时，如果字母表中字母的编码是按顺序来的，就会给我们的工作带来很多便利，显而易见的优点就是，字母的排序和分类将变得简单易行。

幸运的是，这种标准已经存在并且被广泛使用，它被称为美国信息交换标准码

（American Standard Code for Information Interchange），简称为 ASCII 码，发音很像 ASS-key。从 1967 年正式公布至今，它一直是计算机产业中最重要的标准。不过还有一个大的例外（后面会讲到），无论何时，当你在计算机上处理文本时，总会在不经意间使用到 ASCII 码。

ASCII 码是 7 位编码，它的二进制取值范围为 0000000～1111111，对应于十六进制就是 00h～7Fh。现在我们一起来讨论下 ASCII 码，但我不建议从开始学起，因为相对于后面的编码，前 32 个编码理解起来还有一点难度。所以我们从第 2 组 32 个编码开始学习，它包括标点符号和 10 个数字。下表列出了这 32 个字符及相应的十六进制编码。

十六进制编码	ASCII 字符	十六进制编码	ASCII 字符
20	空格	30	0
21	!	31	1
22	“	32	2
23	#	33	3
24	$	34	4
25	%	35	5
26	&	36	6
27	‘	37	7
28	(	38	8
29	)	39	9
2A	*	3A	:
2B	+	3B	;
2C	,	3C	<
2D	-	3D	=
2E	.	3E	>
2F	/	3F	?

值得注意的是 20h 代表空格符，它的作用是将单词或句子隔开。

接下来的 32 个编码是大写字母和一些附加的标点符号的编码。除了@符号和下画线之外，其余的符号很难在打字机上找到。它们真正出现的地方是标准计算机键盘，下表列出了这些字符及相应的十六进制编码。

十六进制编码	ASCII 字符	十六进制编码	ASCII 字符
40	@	50	P
41	A	51	Q

续表

42	B	52	R
43	C	53	S
44	D	54	T
45	E	55	U
46	F	56	V
47	G	57	W
48	H	58	X
49	I	59	Y
4A	J	5A	Z
4B	K	5B	[
4C	L	5C	\
4D	M	5D	]
4E	N	5E	^
4F	O	5F	_

再接下来的 32 个编码是所有小写字母和一些附加的标点符号及其对应的十六进制编码，这些字符也很少在打字机上出现。

十六进制编码	ASCII 字符	十六进制编码	ASCII 字符
60	`	70	p
61	a	71	q
62	b	72	r
63	c	73	s
64	d	74	t
65	e	75	u
66	f	76	v
67	g	77	w
68	h	78	x
69	i	79	y
6A	j	7A	z
6B	k	7B	{
6C	i	7C	|
6D	m	7D	}
6E	n	7E	~
6F	o		

注意，表的最后不包括 **7Fh** 及其对应的字符。如果你统计一下，就会发现这三张表共涵盖了 95 个字符。由于 ASCII 码的编码长度为 7 位，所以最多可以表示 128 个编码，

这样算下来还剩 33 个编码可用。下面我们通过几个简短的例子学习一下编码。

像这样一段字符串：

Hello, you!

转换成 ASCII 码，用十六进制数表示如下：

48 65 6C 6F 2C 20 79 6F 75 21

这段编码中，除了普通的字符，逗号（编码 2C）、空格（编码 20）和感叹号（编码 21）容易遗漏，需要额外注意。我们再来看一个例子：

I am 12 years old.

它用 ASCII 码表示为：

49 20 61 6D 20 31 32 20 79 65 61 72 73 20 6F 6C 64 2E

有意思的是数字 12 的表示方法。在这段编码串中，它被表示成十六进制数 31h 和 32h，也就是数字 1 和 2 的 ASCII 码的组合。当数字 12 以文本流的身份出现时，不应该用十六进制码 01h 和 02h，或者 BCD 码 12h，或者 0Ch 来表示。因为这些编码在 ASCII 码中表示其他的意思。

在 ASCII 码中，一个大写字母与其对应的小写字母的 ASCII 码值相差 20h。这种规律大大简化了程序代码的编写，例如一段将特定的字符串变成大写的程序。假设有一个字符串存放在内存的某个区域，每个字符占据一个字节。下面是一段 8080 子程序，初始状态下字符串的首地址存放在寄存器 HL 中；寄存器 C 存放字符串的长度，也就是字符的个数。

```
Capitalize: MOV A, C        ; c 表示剩余的字符数
            CPI A, 00h      ; 与 0 进行比较
            JZ AllDone      ; 如果剩余的字符数为 0，程序结束
            MOV A, [HL]     ; 取得下一个字符
            CPI A, 61h      ; 判断 A 代表的字符的 ASCII 码是否比'a'小
            JC Skipit       ; 如果比'a'小，就跳过
            CPI A, 78h      ; 判断是否比'z'大
            JNC SkipIt      ; 如果是，则跳过
            SBI A, 20h      ; 判断是否是小写，如果是，则减 20h
            MOV [HL], A     ; 保存修改过的字符
SkipIt:     INX HL          ; 指向下一个字符
            DCR C           ; 计数器减一
```

```
            JMP Capitalize   ; 返回到程序起始处
AllDone:    RET
```

还有另外一种方法也可以将小写字母减去 20h 而转换成大写字母，如下所示：

```
ANI A, DFh
```

ANI 指令（AND Immediate）用来“与”一个立即数。在上面这个例子中，累加器中的数值与 DFh 执行“按位与”操作，其中 DFh 转换成二进制数就是 11011111。“按位与”操作就是把两个数分别转换成二进制，然后将对应的位进行“与”操作。这个例子中，除了自左向右数的第 3 位被置成 0 外，A 中的其他位均被保留。通过将这一位设置为 0，我们实现了将小写字母的 ASCII 码转换成大写字母的目的。

十六进制编码	缩　写	控制字符的含义
00	NUL	空字符
01	SOH	标题开始
02	STX	文本开始
03	ETX	文本结束
04	EOT	传输中止
05	ENQ	询问
06	ACK	应答
07	BEL	响铃
08	BS	回退
09	HT	水平制表
0A	LF	换行
0B	VT	垂直制表
0C	FF	换页
0D	CR	回车
0E	SO	移出
0F	SI	移入
10	DLE	转义
11	DC1	设备控制 1
12	DC2	设备控制 2
13	DC3	设备控制 3
14	DC4	设备控制 4
15	NAK	否定应答
16	SYN	同步

续表

17	ETB	块传输结束
18	CAN	取消
19	EM	媒介取消
1A	SUB	替代字符
1B	ESC	跳出
1C	FS	文件分割或信息分割 4
1D	GS	组分割或信息分割 3
1E	RS	记录分割或信息分割 2
1F	US	单元分割或信息分割 1
7F	DEL	删除

前面讲到的 95 个编码也被称为图形文字（graphic characters），因为它们可以被显示出来。其实 ASCII 码还包含 33 个控制字符（control characters），它们用来执行某一特定功能，因而不用显示出来。为了完整地讨论 ASCII 编码，下面将这 33 个控制字符也列举了出来，有一些的确很难理解，不过不用在意。其实在 ASCII 码公布以后，当时人们更多的是想把它用在电传打字机上，所以，如今其中的许多编码已经渐渐离开了人们的视线。

人们最初的想法是可以在图形字符中使用控制字符，以便对文本格式进行基本的调整。举个例子或许会帮助你更好地理解这种做法：假如有一台电传打字机或者打印机，它负责解析 ASCII 码，解析之后做出相应的操作，最后在纸上打印出字符。设备的打印头每打印一个字符就向右移动一格，通过这种方式来对 ASCII 码做出响应。而要实现这些操作就需要用到控制字符。

举个例子来讲，看看下面这个十六进制字符串：

41 09 42 09 43 09

编码 09 代表水平制表符，简写为 Tab。假设打印的过程中，所有水平排列字符的起始位置都为 0，Tab 的作用是在下一个水平位置即在距前一个字符的间距为字符长度 8 倍的位置打印下一个字符，如下所示：

```
A       B       C
```

这种简单有效方法使得字符可以保持按列对齐。

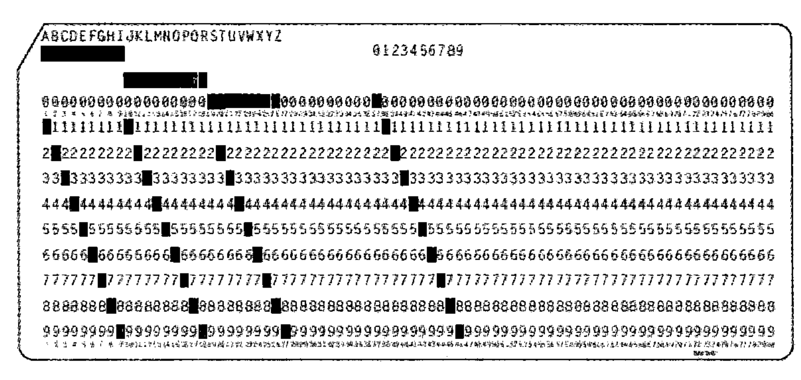
有一些控制字符甚至沿用至今，例如换页符（12h），它使得打印机跳出当前页，并

开始准备打印下一页。

回退符可以用来打印复合字符，尤其是在一些旧的打印机上。假设计算机要控制电传打字机，使其不仅打印小写字母 e，还要将其重音标记出来，即 è。我们可以使用回退符来实现，十六进制码为：65 08 60。

计算机中的回车和换行与 Baudot 码中表示的意思相同，它们可以算得上是控制符中最重要的两个符号。在打印机中，回车符使得打印头换行并转移至当前页面的最左端，换行符使打印头转移至当前位置下一行。这两种操作都使得打印头移至新的一行。回车符通常用来另起一行继续打印，换行符通常在不需要移到页面最左端而换行时使用。

尽管 ASCII 码在计算机领域可以说是一统江湖，但许多 IBM 大型机上却没有采用这种标准。例如，System /360 产品内部采用的是 IBM 自发研制的 8 位字符编码系统，也被称为扩展的 BCD 交换码（Extended BCD Interchange Code），或 EBCDIC（英文中的发音为 EBB-see-dick）。EBCDIC 是早期的 6 位 BCDIC 编码的扩展形式，BCDIC 的起源于 IBM 的打孔卡。一张打孔卡——存储容量为 80 个文本字符——1928 年由 IBM 首创并沿用了将近 50 年，它的外观如下图所示。



在考虑打孔卡与 8 位 EBCDIC 字符码的关系时，需要知道，在几代技术的影响下，这种编码也历经几十年的演变。因此，打孔卡与 EBCDIC 之间的逻辑性和一致性也逐渐消失了。

打孔卡上每一列穿出的一个或多个矩形孔代表一个字符，而这些字符一般也打印在卡片的顶部。最下面的 10 行由数字标识，自上向下分别为第 0 行、第 1 行直到第 9 行。第 0 行上面的一行通常不出现数字，称为第 11 行，顶端为第 12 行，这里没有第 10 行。

以下面列举一些 IBM 打孔卡的常用术语：第 0～9 行称做数字行（digit rows）或数字

穿孔(digit punches),第 11 和 12 行被称做区域行(zone rows)或区域穿孔(zone punches)。由于不统一，IBM 打孔卡用起来有时会有些混乱：比如有的卡片把第 0 和第 9 行看做是区域行而不是数字行。

一个 EBCDIC 字符码由 8 位比特组成，进一步可以细分为高半字节(4 比特)与低半字节。低半字节是 BCD 码，与字符的数字穿孔保持一致，高半字节与区域穿孔的编码保持一致(而且与区域穿孔一一对应)。回忆一下第 19 章的 BCD 编码原理，其本质是采用二进制数对十进制数进行编码(binary-coded decimal)——其中数字 0~9 都利用不同的 4 位二进制数进行编码。

数字 0~9 并不需要区域穿孔进行额外表示，它们的 EBCDIC 编码的高半字节是 1111，代表了区域穿孔不起作用，而 0~9 的 EBCDIC 编码的低半字节是数字穿孔的 BCD 码，如下所示。

十六进制编码	EBCDIC 字符
F0	0
F1	1
F2	2
F3	3
F4	4
F5	5
F6	6
F7	7
F8	8
F9	9

大写字母有一些有趣的规律，如果区域穿孔只出现在第 12 行，则高半字节标识为 1100；如果只出现在第 11 行，则高半字节标识为 1101；如果出现在第 0 行，则高半字节标识 1110。下表给出了大写字母及其对应的 EBCDIC 编码。

十六进制编码	EBCDIC 字符	十六进制编码	EBCDIC 字符	十六进制编码	EBCDIC 字符
C1	A	D1	J		
C2	B	D2	K	E2	S
C3	C	D3	L	E3	T
C4	D	D4	M	E4	U
C5	E	D5	N	E5	V
C6	F	D6	O	E6	W

续表

C7	G	D7	P	E7	X
C8	H	D8	Q	E8	Y
C9	I	D9	R	E9	Z

值得注意的是 R 与 S 之间编号有跳变。有时在编写程序的时候，尤其是程序中用到 EBCDIC 编码时，这个容易被忽视的小细节往往会令人抓狂。

小写与大写字母的数字穿孔是相同的，但它们的区域穿孔不同。在 a～i 的小写字母，穿孔位于第 12 行和第 0 行，高半字节对应的编码为 1000；在 j～r 的小写字母，穿孔位于第 12 行和第 11 行，高半字节对应的编码为 1001；在 s～z 的小写字母，穿孔位于第 11 行和第 0 行，高半字节对应的编码为 1010。小写字母的 EBCDIC 字符及其对应的十六进制编码如下表所示。

十六进制编码	EBCDIC 字符	十六进制编码	EBCDIC 字符	十六进制编码	EBCDIC 字符
81	a	91	j		
82	b	92	k	A2	s
83	c	93	l	A3	t
84	d	94	m	A4	u
85	e	95	n	A5	v
86	f	96	o	A6	w
87	g	97	p	A7	x
88	h	98	q	A8	y
89	i	99	r	A9	z

当然，标点符号和控制字符也都有自己的 EBCDIC 编码，但对于这些字符的编码系统没有必要去深究。

仔细观察 IBM 打孔卡，其中每一列细细数下共有 12 个孔，每个孔代表 1 位，也就是说可以提供 12 位的编码信息，不是吗？我们其实可以用打孔卡上每一列 12 孔中的 7 个来表示 ASCII 码。但是，这种方案有一个非技术方面的缺陷，那就是太多的穿孔将使得卡片变得很脆弱，容易折断。

采用 8 位编码的 EBCDIC 中其实还有很多编码未定义，这也说明当年 ASCII 码采用了 7 位编码也是合乎情理的。在 ASCII 码刚刚问世的那个年代，存储器的价格贵得令人咋舌，有一些观点认为 ASCII 码可以用 6 位编码并配合转义字符来使用，这样既可以区

分大小写又节约了存储器。这种方案并没有被采纳，当时还有一些人认为 ASCII 码应采用 8 位编码，他们对计算机的体系结构有了一个大胆的推测，即计算机应该按字节存储，7 位存储是不合适的。今天来看，8 位的字节存储已经作为了一项标准。尽管 ASCII 码从技术的本质上来看是 7 位编码，但仍以 8 位的形式存储。

在字节与字符之间建立一种等价关系大大简化了我们的工作，举例来讲，如果要粗略估计一个文本文件所需要的存储空间，只要统计字符数就可以了。这时前面学过的 K（kilos）和 M （Megas）就派上了用场，用它们来表示文本所占据的计算机存储空间更加通俗易懂。

传统的排版格式是：一张大小为 8.5×11 英寸的打印纸，采用双倍行距，1 英寸的页边距，每页可以容纳约 27 行的正文。每行宽度约为 6.5 英寸，每英寸可容纳 10 个字符，通过计算可以知道每页共包含约 1750 个字节。如果页面采用单倍行距，那么打印纸的容量约为原先的 2 倍，即 3.5 KB。

翻开一本《纽约客》（*The New Yorker*）杂志，可以看到杂志每页有 3 栏，每栏包含 60 行，每行约有 40 个字符，这样算下来每页大致包含 7200 个字符（也可以说成字节）。

《纽约时报》（*New York Times*）每一页包含 6 栏。假如页面都是文字而不包含标题和图片（这其实是不大可能的），那么可以认为每栏包含 155 行，每行大约容纳 35 个字符，这样算下来整个页面共包含 32,550 个字符，即 32 KB。

一般来讲精装书每页大约包含 500 个单词。根据统计，每个单词平均占用 5 个字母——更确切地来讲应该是 6 个字母，因为单词与单词之间是通过空格来分隔的，所以要一并统计在内。这样算下来，书的每一页大约包含 3000 个字符。假设每本书平均页数为 333，这个估计或许和实际不符，但如果这样估算的话，每本书平均容量为 1 MB。

不得不承认的是，书与书之间千差万别，所以上面这些也只是估算，下面列举出一些实际数据。

斯科特・菲茨杰拉德（F. Scott Fitzgerald）的《了不起的盖茨比》（*The Great Gatsby*）大约 300 KB。

塞林格（J. D. Salinger）的《麦田守望者》（*Catcher in the Rye*）大约 400 KB。

马克・吐温（Mark Twain）的《哈克贝里・弗恩历险记》（*The Adventures of Huckleberry*

Finn）大约 540 KB。

约翰·斯坦贝克（John Steinbeck） 的《愤怒的葡萄 / 怒火之花》（*The Grapes of Wrath*）大约 1MB。

赫尔曼·梅尔维尔（Herman Melville）的《白鲸》（*Moby Dick*）大约 1.3 MB。

亨利·菲尔丁（Henry Fielding）的《弃儿汤姆·琼斯的历史》（*The History of Tom Jones*）大约 2.25 MB。

玛格丽特·米切尔（Margaret Mitchell）的《乱世佳人》（*Gone With the Wind*）大约 2.5 MB。

斯蒂芬·金（Stephen King）的《末日逼近》（*The Stand*）大约 2.7 MB。

列夫·托尔斯泰（Leo Tolstoy）的《战争与和平》（*War and Peace*）大约 3.9 MB。

马塞尔·普鲁斯特（Marcel Proust）的《追忆似水年华》（*Remembrance of Things Post*）大约 7.7 MB。

美国国会图书馆（The United States Library of Congress）藏书约为 2000 万本，大概有 20 万亿字符，从存储器角度来说，数据总量为 20 TB（这还不包括图书馆中的大量珍贵照片和录音资料）。

尽管 ASCII 码是计算机领域最重要的标准，但它并不是十全十美的。它的问题就蕴含在它的全称中——American Standard Code for Information Interchange，它是太美国化了！即使那些以英语为主要语言的国家，ASCII 码也并不适用。ASCII 码中包含美元符号，而英镑符号怎么找不到呢？还有西欧国家语言中用到的重音符号在哪里？更别说使用非拉丁字母的希腊文（Greek）、阿拉伯文（Arabic）、希伯来文（Hebrew）和西里尔文（Cyrillic）等欧洲国家了。此外，印度及东南亚地区用到的婆罗门手记、北印度的 Devanagari 方言、孟加拉语、泰语、西藏语也并没有在 ASCII 码中出现。简单的 7 位编码在面对数以万计的中国、日本、韩国的象形文字，以及奇怪的朝鲜文音节时也显得力不从心。

在 ASCII 码的发展历程中，尽管没有在引入非拉丁字母方面做过工作，但开发者也一直在积极思考与改进编码系统，使其适用于其他国家。根据公布的 ASCII 码标准，有 10 个 ASCII 码保留位（40h、5Bh、5Ch、5Dh、5Eh、60h、7Bh、7Ch、7Dh 和 7Eh）可被重新定

义，这样就便于特定国家的使用。另外，英镑符号（£）可以在需要时替换特殊符号（#），通用货币符号（¤）可以在需要时替换美元符号（$）。当然，为使得这一替换过程不发生混淆，如果在文本文件中使用了这些重定义的符号，相关人员都必须知道这些变化。

大多数计算机系统采用 8 位编码来存储字符，我们也自然地想到设计一种扩展的 ASCII 字符集，这样可以包含 256 个字符，比原先扩展了一倍。在这种字符集中，编码 00h～7Fh 与原 ASCII 码保持一致；编码 80h～FFh 可以用来引入其他字符。这项技术已经被用来定义附加的字符编码，比如前面提到过的重音字母以及非拉丁字母。下面这个例子是对 96 个额外字符的 ASCII 码扩展，称为第 1 号拉丁字母表（Latin Alphabet No. 1），其中包括 A0h～FFh 字符编码。在该表中，每个字符的十六进制编码的高半字节由第一行给出，低半字节由第一列给出，如下表所示。

	A-	B-	C-	D-	E-	F-
-0		°	À	Ð	à	ð
-1	¡	±	Á	Ñ	á	ñ
-2	¢	²	Â	Ò	â	ò
-3	£	³	Ã	Ó	ã	ó
-4	¤	´	Ä	Ô	ä	ô
-5	¥	µ	Å	Õ	å	õ
-6	¦	¶	Æ	Ö	æ	ö
-7	§	·	Ç	×	ç	÷
-8	¨	¸	È	Ø	è	ø
-9	©	¹	É	Ù	é	ù
-A	ª	º	Ê	Ú	ê	ú
-B	«	»	Ë	Û	ë	û
-C	¬	¼	Ì	Ü	ì	ü
-D	-	½	Í	Ý	í	ý
-E	®	¾	Î	Þ	î	þ
-F	¯	¿	Ï	ß	ï	ÿ

编码 A0h 对应的字符为不中断空格（No-Break Space）。通常计算机在对文本进行排版时，会将其划分为行和段，行与行之间以空格符号区分（空格所对应的 ASCII 码为 20h）。编码 A0h 显示为空格，但是并不表示行与行之间被断开。比如在“WW II”这样一段文

字中就可以使用不中断空格。编码 ADh 被定义为软连字符（soft hyphen），它的用途是连接同一单词之间的音节，在一个单词被不得已划分在两行时就会用到它。

只可惜问题也随之而来，近几十年来出现了许多不同版本的扩展的 ASCII 码，多个不同的版本严重影响了编码的一致性，导致了混淆和不兼容。ASCII 码被扩展到极致，有的甚至可以对中文、日文和韩文进行编码。其中有一种流行的编码——Shift-JIS，即日本工业标准（Japanese Industrial Standard），利用 81h～9Fh 表示双字节字符编码的初始字节。通过这种手段，Shift -JIS 可对额外的约 6000 个字符进行编码。只可惜 Shift-JIS 并不是唯一的采用这种技术的编码系统。在亚洲地区，还有三个类似的双字节字符编码系统（double-byte character sets，DBCS）同样也很流行。

双字节字符集的确有很多版本，但兼容性并不是它最主要的问题。它的另一个缺陷是，一些字符，特别是通用的 ASCII 码字符，是用单个字节编码表示的，相比而言，成千上万的象形文字则是双字节编码，这在无形之中增加了使用这种字符集的难度。

业界一直有一个目标，那就是建立一个独一无二的字符编码系统，它可以用于世界上所有语言文字，从 1988 年开始，几大著名计算机公司合作研究出一种用来替代 ASCII 码的编码系统，取名为 Unicode（统一化字符编码标准）。相对于 ASCII 的 7 位编码，Unicode 采用了 16 位编码，每一个字符需要 2 个字节。也就是说 Unicode 的字符编码范围为 0000h～FFFFh，总共可以表示 65,536 个不同字符。全世界所有的人类语言，尤其是经常出现在计算机通信过程中的语言，都可以使用同一个编码系统，而且这种系统还具备很高的扩展性。

Unicode 编码其实并不是从零开始设计的，前 128 个字符编码——即 0000h～007Fh——与 ASCII 码是一致的。Unicode 编码中的 00A0h～00FFh 与先前讲到的第 1 号拉丁字母表是一致的。全世界很多标准也被一同收录在 Unicode 中。

尽管相对于之前讲过的一些字符编码系统，可以说 Unicode 做出了有效地改进，但这也不能确保它被全世界广泛采纳。ASCII 码，包括数不清的有一点小缺陷的扩展 ASCII 码已经在计算机领域根深蒂固，想一下子就取代它们并不是轻而易举的。

对于 Unicode 来讲，它唯一的问题，就是它改变了字符与存储空间之间“单字符，单字节”的等价对应关系。采用 ASCII 编码方式存储的著作《怒火之花》，其所占据的存储空间约为 1 MB。而如果采用 Unicode 编码，约占 2 MB。为了使编码系统兼容，Unicode 在存储空间上付出了相应的代价。

21

总　线

在一台计算机中，中央处理器无疑是最重要的部件，但它并不是唯一的部件。随机访问存储器（Random Access Memory，RAM）也是计算机不可或缺的部件，它存放着处理器要执行的机器代码指令。通过怎样的方法才能把指令加载到 RAM 中？怎样才能把程序的结果变得可见呢？或许你一下子就想到了输入设备(Input Device)和输出设备(Output Device)。回想一下前面讲过的内容，RAM 是易失性存储器——换言之，当掉电的时候其中的内容就会丢失。所以，长期存储设备也是一台计算机必不可少的部件，只有这样，代码和数据才能够被永久保存，不会因为掉电而丢失重要的数据。

搭建一台完整的计算机还需要很多集成电路，这些集成电路都必须挂载（mounted）到电路板上。在一些小型的机器中，一块电路板足以容纳所有的集成电路，但这种情况并不常见。我们通常所看到的是另一种情况：计算机中各部件按照功能被分别安装在两个或更多的电路板上。这些电路板之间通过总线（bus）通信。如果对总线做一个简单的概括，可以认为总线就是数字信号的集合，而这些信号被提供给计算机上的每块电路板。通常把这些信号划分为如下四类。

- 地址信号。这些信号是由微处理器产生，通常用来对 RAM 进行寻址操作，当然也可以用来对连接到计算机的其他设备进行寻址操作。

- 数据输出信号。这些信号也是由微处理器产生的，用来把数据写入到 RAM 或其他设备。这里特别要注意区分术语输入（input）和输出（output），来自微处理器的数据输出信号会变成 RAM 和其他设备的数据输入信号。
- 数据输入信号。这些信号是由计算机的其他部分提供的，并由微处理器读取。通常情况下，数据输入信号由 RAM 输出，这就解释了微处理器是怎样从内存中读取内容的。其实，其他部件也可以给微处理器提供数据输入信号。
- 控制信号。这些信号是多种多样的，通常与计算机内所用的特定的微处理器相对应。控制信号可以产生于微处理器，也可以由与微处理器通信的其他设备产生。比如，当微处理器要把一些数据写入到特定内存单元时，它所使用的信号就是控制信号。

还有一点需要说明：总线还可以为计算机上不同电路板供电。

回顾一下总线的发展历程。在家用计算机领域，早期比较流行的就是 S-100 总线，1975 年第一台家用计算机 MITS Altair 就率先采用了这种总线。尽管一开始，S-100 总线只是基于 8080 微处理器的，后来经过改进，也开始适用于其他处理器，例如 6800。一块 S-100 电路板的规格是 5.3×10 英寸，其中有一边是要插到一个插槽上的，这个插槽有 100 个连接器（这就是名为 S-100 的原因）。

每台 S-100 计算机都有一块很大的被称为母板（motherboard 或 mainboard）的电路板，它有若干相互连接的 S-100 总线插槽（可能有 12 个）。有时候，这些插槽也被称为扩展插槽（expansion slots）。S-100 电路板（也称为扩展板，expansion boards）就插在这些插槽中。8080 微处理器及支持芯片（第 19 章提到过的其中的一些）分布在一块 S-100 电路板上，而 RAM 分布在一块或多块其他电路板上。

S-100 总线是专门为 8080 芯片而设计的，有 16 个地址信号，8 个数据输入信号及 8 个数据输出信号（仔细回忆一下，8080 本身并不区分数据输入和输出信号，这项工作是由电路板上的其他支持芯片完成的）。总线上也含有 8 个中断信号，其他设备需要 CPU 立即做出响应时，便会产生这些信号。下面我们看一个例子（本章的后面也要讲到），当某个按键按下时，键盘可能就会产生一个中断信号。接下来 8080 会执行一段小程序，检测出是什么按键被按下，并做出响应。通常，在安装了 8080 的电路板上有一个被称为 Intel 8214 优先级中断控制单元的芯片，就是专门用来处理中断的。当中断发生时，这个芯片

会产生一个中断信号并送给 8080。8080 识别出这个中断后，此芯片就会提供一个 RST（Restart，重启）指令，在这条指令的作用下，微处理器会把当前程序计数器的值保存下来，并依据中断类型，跳转到地址 0000h、0008h、0010h、0018h、0020h、0028h、0030h 或 0038h 处执行。

如果你在设计一个新的计算机系统，而这个系统中采用新的总线类型，你可以选择把总线规范公布于众（也可以通过其他方式发布出去）或者使其保密，决定权在于你。

一旦一条指定总线的规范公布开来，其他制造商——称为第三方（third-party）制造商——就可以设计并销售采用了这种总线的扩展板了。这些额外扩展板不仅加强了计算机的实用性，还使其更加满足实际需求。计算机的销售情况越好，扩展板的市场前景也就越好。正是由于这个原因，设计者在设计多数小型计算机系统时，都会坚持开放体系结构（Open Architecture）的原则，这样一来，其他制造商就可以生产计算机的外设。最终会有一条总线成为工业标准。在今天，“标准”已经成为个人计算机产业的一个重要组成部分。

1981 年秋，最著名的开放体系结构个人计算机——IBM 的 PC 问世。IBM 公布了 PC 的技术参考资料（technical reference），里面包含了整台计算机的全部电路图，IBM 为其制造的扩展板的资料也在其中。这个手册可是很重要的资料，它的出现使得很多制造商可以生产自己的 PC 扩展板，实际上，这创造出了整个 PC 的克隆体——其实与 IBM 的 PC 几乎完全相同，运行的软件也一样。

在如今的桌面计算机领域，从起初的 IBM 的 PC 发展而来的计算机数量庞大，占据约 90%的市场份额。尽管 IBM 本身只占很小一部分，但事实上，如果起初的 PC 采用的是封闭体系结构（closed architecture）且设计是私有化（proprietary）的，其所占的市场份额会更少。苹果公司的麦金托什（Macintosh）起初采用的是封闭体系结构，尽管也曾部分考虑过开放的问题，这也就解释了为什么 Macintosh 在如今的桌面计算机市场上只占有不到 10%的份额。（请记住这一点：一个计算机系统可以是在开放体系结构下设计的，也可以是在封闭体系结构下设计的，无论是哪种情况，其他公司都可以为其设计软件。但也有例外的情况，某些视频游戏的开发商会限制其他公司开发其专用系统上的软件）。

最初的 IBM PC 采用的是 Intel 8088 微处理器，可以寻址 1 MB 的存储单元。虽然 8088 内部是一个 16 位的微处理器，但外部却只能寻址 8 位的存储器。工业标准体系结构

（Industry Standard Architecture，ISA）总线，是 IBM 为最初的 PC 设计的。扩展板上有 62 针的连接插头。有 20 个地址信号，8 个复用的数据输入/输出信号，6 个中断请求信号及 3 个直接存储器访问（Direct Memory Access，DMA）请求信号。DMA 可以使存储设备（本章最后我们会讲到）快速地执行存储操作，这比采用其他方法快得多。通常情况下，所有读/写内存的操作都是由微处理器来完成的，但采用了 DMA 后，其他设备可以不通过微处理器而获得总线的控制权，进而直接对内存进行读写。

在 S-100 系统中，所有的部件都安放在扩展板上。就拿 IBM PC 来说，微处理器、支持芯片及一些 RAM 都安装在一块系统板上，系统板（system board）是 IBM 的“内部称呼”，但它常常也被称为母板或主板。

1984 年，IBM 推出了个人计算机 AT，采用的是 16 位的 Intel 80286 微处理器，这个微处理器可以寻址 16 MB 的存储单元。IBM 保留了原有的总线，但添加了一个 36 针的插槽。这个插槽包括 7 个地址信号（其实只需 4 个）、8 个数据输入/输出信号、5 个中断请求信号以及 4 个 DMA 请求信号，这些信号都是新增的。

微处理器所使用的数据宽度（从 8 位到 16 位再到 32 位）和输出的地址信号的数目在不断增长，当这些超出总线的承受能力时，总线就需要升级换代了。如果微处理器的处理速度很快时，也会出现这种情况。早期的总线是为当时的微处理器而设计的，它们的时钟频率一般是几兆赫兹而不是几百兆赫兹。如果设计出来的总线不适合高速传输的话，就会出现射频干扰（RFI），这会使附近的收音机和电视机产生静电或其他噪声。

1987 年，IBM 推出了微通道体系结构（Micro Channel Architecture，MCA）总线。这种总线的某些部分已经成为 IBM 的专利，如果其他公司使用这种总线，IBM 就会从中收取授权费用。也许就正是由于这个原因，MCA 才没能成为一种工业标准。然而就在 1988 年，9 家公司（并不包括 IBM）联合推出的 32 位 EISA（Extended Industry Standard Architecture）总线取代了 MAC，成为了工业标准。近几年，Intel 公司设计的外围部件互连（PCI）总线已普遍使用在 PC 兼容机上。

计算机上的各种不同的部件是如何工作的呢？为了能更好地理解，让我们再次回到 20 世纪 70 年代中期去看一看。想象一下，我们正在为 Altair 设计电路板，或者是在为自己设计的 8080 或 6800 计算机做这样的事情。我们不仅要考虑为计算机设计一些存储器，用键盘作为输入，用电视机作为输出；还要考虑关上计算机时，如何把存储器中的内容

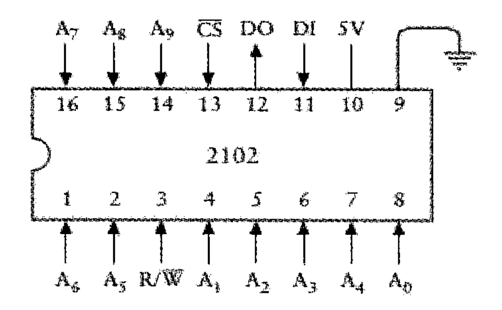
保存下来。如何把这些部件添加到计算机中呢？下面就来看看能实现这个功能的各种接口（Interface）。

现在回想一下第 16 章所讲的内容，RAM 阵列有地址输入、数据输入，以及数据输出信号，另外还有一个用来把数据写入存储器的控制信号。RAM 阵列能存放的数据的数量是和地址输入信号的个数有关的，它们之间有着如下的关系：

$$\text{RAM 阵列中数字的个数} = 2^{\text{地址信号的个数}}$$

讲到这里你可能会问，数据输入、输出信号又有怎样的作用呢？其实它们决定着所存储的数值的大小（位数）。

20 世纪 70 年代中期，2102 是用于家用计算机的一款流行的存储器芯片。其管脚分布如下图所示。



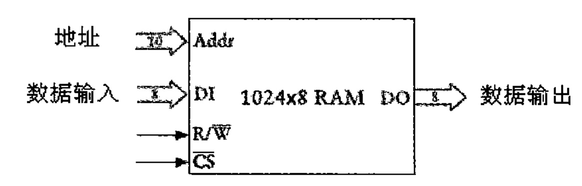
它也是 MOS（metal-oxide semiconductor）家族中的一员，与 8080 和 6800 微处理器所采用的技术相同。MOS 半导体管很容易与 TTL 芯片连接起来；通常情况下，其内部晶体管的密度要比 TTL 高，但速度却不如 TTL 快。

这个芯片存储容量可以达到 1024 位，这个数值可以根据地址信号（A0 ~ A9）、数据输出（DO）和数据输入（DI）信号（输入和输出复用一个信号线）的数目计算出来。你所使用 2102 芯片型号不同，访问时间（read access time，指从芯片接收到地址信息到输出有效数据所需的时间）也是各有差异，从 350 ns ~ 1000 ns 不等。当需要从存储器中读取数据时，$R/\overline{W}$（读/写）信号置 1；当向芯片中写入数据的时候，这个信号要置 0，而且至少要持续 170 ~ 550 ns 的时间，也是由所使用的 2102 芯片的型号决定的。

这里我们不得不提到的一个信号就是 $\overline{CS}$ 信号，也称片选信号。该信号置 1 时，芯片不被选中，意思就是说，不会响应 $R/\overline{W}$ 信号。其实，$\overline{CS}$ 信号的作用不止这些，对芯片

还有其他重要的作用，下面我们将简单描述一下。

想想看，若让你为 8 位的微处理器组织存储器的话，你会怎么做呢？是选择按 8 位存储形式，还是 1 位存储形式？你肯定会选择前者。如果想存储整个字节，则至少需要 8 个这样的 2102 芯片。具体的做法就是，把 8 个芯片对应的地址信号、$R/\overline{W}$ 及 $\overline{CS}$ 信号连接起来，如下图所示。



实际上，这是一个 1024×8 位的 RAM 阵列，或者说是容量为 1 KB 的 RAM。

把存储器芯片安装在一块电路板上，这是很符合实际的做法。那么，到底一块电路板上能安装多少块这样的芯片呢？如果是紧紧排列在一起的话，一块 S-100 板就能容纳 64 个。这样一来，就提供了一个 8 KB 的存储空间。一般我们不这样做，更合适的方法是，用 32 个芯片组成一个 4 KB 的存储器。为了存储完整的字节，而连接在一起的芯片的集合，称为存储体（bank）。例如，一个 4 KB 大小的存储器板就由 4 个存储体组成，而每个存储体又包含 8 个芯片。

8 位微处理器，例如 8080、6800，有 16 位地址，可用来寻址 64 KB 的存储空间。如果你制作了一个包含 4 个存储体、大小为 4 KB 的存储器板，则存储器板上的 16 位地址信号就有如下所示的功能。

$A_{15}\ A_{14}\ A_{13}\ A_{12}\ A_{11}\ A_{10}\ A_9\ A_8\ A_7\ A_6\ A_5\ A_4\ A_3\ A_2\ A_1\ A_0$

选择存储器板　选择存储体　寻址 RAM

下面详细解释一下这 16 位地址信号。A0～A9 直接与 RAM 芯片相连接；A10 和 A11 用来选择 4 个存储体中要被寻址的那一个；A12～A15 确定哪些地址申请用这块存储器板，换言之，就是这块存储器板响应哪些地址。微处理器整个存储空间的大小是 64 KB，被划分成 16 个不同的区域，每个区域的大小是 4 KB，我们设计的 4KB 存储器板占用了其中一个区域。这 16 个区域划分情况如下。

0000h～0FFFh

1000h～1FFFh

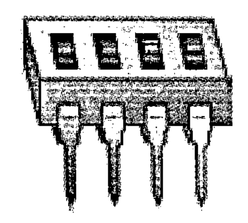
2000h～2FFFh

……

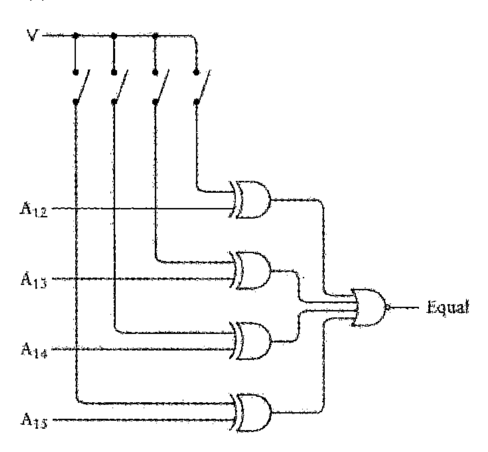
F000h～FFFFh

举例说明，假定 4 KB 存储器板使用了 A000h～AFFFh 地址区域。这就意味着，第一个存储体占用了地址 A000h～A3FFh，第二个占用了地址 A400h～A700h，第三个占用了地址 A800h～ABFFh，剩下的 AC00h～AFFFh 地址空间分给了第四个存储体。

你完全可以制作一块 4 KB 存储器板，在用到它的时候再灵活确定其地址范围。要获得这样的灵活性，可以使用一种名为双列直插式封装（dual inline package，DIP）开关的器件。在 DIP 中，有一系列极小的开关（从 2 到 12 个不等）。DIP 是可以插在标准的 IC 插槽中的，如下图所示。

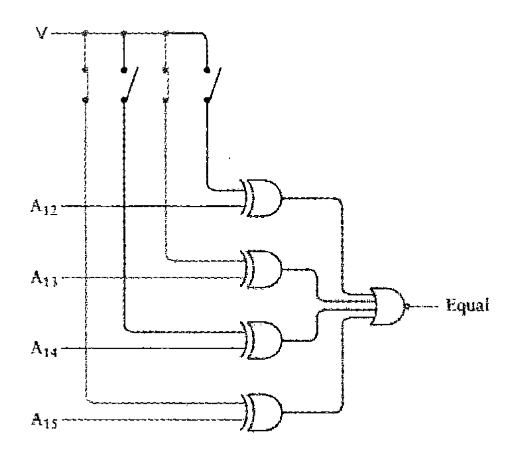


在一种称为比较器（comparator）的电路中，你可以把这个开关和总线上地址信号的高 4 位连接起来，就像下面这样。

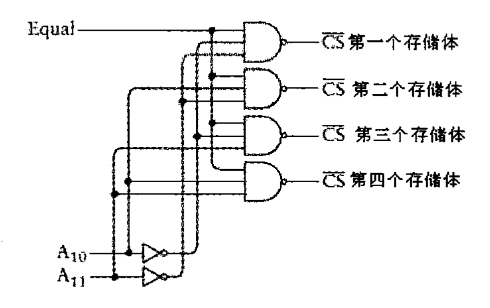


回想一下前面讲过的内容，异或（XOR）门电路在两个输入端中只有一个是高电平时，输出才为高电平；当两个输入端同时为低电平或高电平时，输出是低电平。

例如，如果把 A_{13} 和 A_{15} 对应的开关闭合，就意味着让存储器板能响应存储器空间 A000h ~ AFFFh。若总线上的地址信号 A_{12}、A_{13}、A_{14} 和 A_{15} 与开关上设置的值相同的话，四个异或（XOR）门的输出都是 0，或非（NOR）门的输出为 1，如下图所示。



接下来我们把 Equal 信号和一个 2-4 译码器联合起来使用，就能为四个存储体中的每一个都产生一个 $\overline{\text{CS}}$ 信号，便于对存储体进行选择。具体连接图如下图所示。



例如，若想选择第三个存储体，把 A_{10}、A_{11} 分别置 0 和 1 就可以了。

现在回想一下在第 16 章中阐述过的如何组织 RAM 阵列，这一过程的细节是十分繁琐的，你可能会认为我们还需要 8 个 4-1 选择器，用来从 4 个存储体中选择正确的数据输出信号。但我们并没有这么做，下面来讨论下原因。

通常情况下，TTL 兼容集成电路的输出信号要么大于 2.2V（逻辑 1）要么小于 0.4V（逻辑 0）。试想一下，如果把输出信号连接起来会发生什么呢？一个集成电路的输出为 1，

另一个集成电路的输出为 0，若把这两个输出连接在一起，结果又是什么呢？恐怕谁也无法回答。就是由于这种不确定性，一般不会把集成电路的输出信号连接在一起。

2102 芯片的数据输出信号是三态（tri-state）的，也就是说，除了逻辑 0 和逻辑 1 之外，数据输出信号还有第三种状态。我们必须清楚地认识这种状态——它其实是一种“真空”态，就像芯片的引脚上什么也没连一样。当片选信号（$\overline{CS}$）为 1 的时，2102 芯片的数据输出信号就会进入这种状态。这样一来，我们可以把 4 个存储体相应的数据输出信号连接在一起，并且可以把 8 个输出复用作为总线的 8 个数据输入信号。

之所以强调三态输出的概念，是因为它对总线的操作是至关重要的。几乎所有连接在总线上的器件都使用由总线传递而来的数据输入信号。但不管何时，连接在总线上的电路板中只有一个能确定总线数据输入信号的类型，其他电路板处于三种状态中的无效状态。

或许大家听说过，2102 是一款静态随机访问存储器芯片（Static Random Access Memory，SRAM），它与动态访问存储器（Dynamic Random Access Memory，DRAM）是不同的。通常对于每 1 位存储空间，SRAM 需要用 4 个晶体管（在第 16 章中讲过将触发器作为存储器用，其用到的晶体管更多），而 DRAM 只需要 1 个晶体管，但 DRAM 需要较复杂的外围支持电路，这正是它的缺点。

SRAM 芯片，例如 2102，在电源持续供电的情况下，其内容就能保留下来；一旦掉电，其内容就会丢失。在这方面，DRAM 和 SRAM 很类似。但不同的是，DRAM 芯片在使用时需要定期访问其存储器中的内容，尽管有时并不需要这些内容。这一过程称之为更新（refresh）周期，每秒钟都必须进行几百次。这种做法就好像为不让某人入睡而每隔一段时间就用手肘轻推他一样。

尽管业界在使用 DRAM 上有些争论，但近年来，DRAM 芯片的容量日益增加，使得 DRAM 最终成为标准。1975 年，英特尔公司推出了一款 DRAM 芯片，容量为 16,384 位。其实，DRAM 芯片在容量上基本每三年翻两番，符合摩尔定律。如今，计算机主板上一般都配备内存插槽，这些内存插槽可以容纳几块小存储器板，分为单列直插内存模块（single inline memory modules，SIMM）和双列直插内存模块（dual inline memory module，DIMM）两种，里面包含好几个 DRAM 芯片。如今，花费不到 300 美元就可以买到 128 MB 的 DIMM 了。

既然已经知道如何制作存储器板了，应该没有人会把微处理器的整个存储空间都分配给存储器，必须留些空间给输出设备。

电子射线管（cathode-ray tube，CRT）——20 世纪上半个世纪，在家庭中常见的物件，它从外观上看就像电视机一样——已经成为最常见的计算机输出设备了。我们称连接到计算机上的 CRT 为视频显示器（video display）或监视器（monitor），而称可以为视频显示器提供信号的电子元件为视频适配器（video display adapter）。通常在计算机中，视频适配器是独立存在的，它们拥有自己的电路板，也就是我们常说的显卡（video board）。

表面上看来，视频显示器或电视机的二维图像很复杂，但实际上它是由一束连续的光束射线迅速扫描屏幕而形成的。射线从屏幕左上角开始，从左到右进行扫描，到达屏幕边缘后又折回向左，进行第二行扫描。我们称每一个水平行为扫描行（scan line），称射线回到每个扫描行的开始位置为水平回归（horizontal retrace）。当完成了对最后一行的扫描时，射线不会停下来，它会从屏幕的右下角返回到屏幕的左上角（垂直回归，vertical retrace），并重复上一过程。就拿美国的电视信号来说，每秒钟要进行 60 次（称为场频，field rate）这样的扫描。由于扫描的速度很快，所以不会看到图像出现闪烁的现象。

电视机采用的是隔行（interlaced）扫描技术，情况要复杂些。我们先来看一下帧（frame）的概念，帧是一个完整的静态视频图像，两个场（field）才能形成一个单独的帧。整个帧的扫描线分由两个场来完成——偶数扫描线属于第一个场，奇数扫描线属于第二个场。这里要说明一下水平扫描频率（horizontal scan rate）的概念，即扫描每个水平行的速率，例如 15,750 Hz。把这个数除以 60 Hz，结果是 262.5 行，这正是每个场所包含的扫描线的数目，整个帧的扫描线的数目是场的两倍，也即 525 行。

不管隔行扫描技术是怎样实现的，组成视频图像的连续射线都是由一个连续的信号所控制。虽然一套电视节目的声音和图像部分是一起播出的，但若想把它们广播出去或者通过有线电视系统传送出去，就不得不分开进行。这里所说的视频信号其实与 VCR、录像机、摄像机及一些电视机上的视频输入或输出信号是一样的。

黑白电视机的视频信号十分简单且易于理解（彩色电视机要稍微复杂些）。每秒钟扫描 60 次，扫描信号包含一个垂直同步脉冲（vertical sync pulse），用来指示一个场的开始。这个脉冲为 0 V，宽度约为 400 ms。相比较而言，水平同步脉冲（horizontal sync pulse）则用来指示每个扫描行的开始：视频信号为 0 V，宽度为 5 ms，每秒钟出现 15,750 次。

在两个水平同步脉冲之间，信号的电压是在 0.5～2.0 V 范围内变化的，其中 0.5 V 表示黑色，2.0V 表示是白色，处于两者之间的电压则表示一定的灰度。

正是由于上述原因，电视才会出现部分是数字图像、部分却是模拟图像的情况。虽然在垂直方向上，图像被分为 525 行，但每个扫描行的电压却是连续变化的——用来模拟图像的可视强度。这并不等于说，电压可以随意地变化。事实上，电视机能响应的信号变化频率是有上限的，我们称这一上限为电视机带宽（bandwidth）。

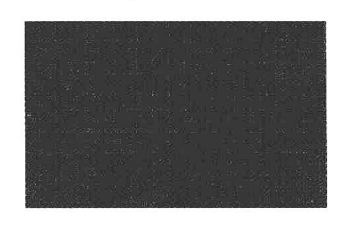
在通信领域中，带宽是极其重要的概念，某个特定的传输媒介能够传输的信息量都是受带宽限制的。以电视机为例，带宽限制了视频信号从黑到白然后又回到黑这一变化的速率。对于美国的广播电视来说，带宽大约为 4.2 MHz。

一旦我们把视频显示器连接到计算机上，就不该把它作为模拟和数字的混合设备来对待，把它看做是完完全全的数字设备更合适一些。从计算机的角度来说，我们可以很方便地把视频图像想象成由离散点组成的矩形网格，这些离散点称为像素（这一术语来自 picture element）。

水平扫描行上像素的个数是受带宽严格限制的。在这里，我把带宽定义为视频信号从黑到白然后又回到黑的变化速率。如果电视机的带宽为 4.2 MHz，它就允许 2 个像素每秒 420 万次的变化，或者——用 2×4,200,000 除以水平扫描速率 15,750——每个水平扫描行有 533 个像素。但并不是所有的像素都可用，约 1/3 的像素被隐藏了起来——处于图像的远端或射线的水平回归中。这样算来，水平扫描行上可用的像素约为 320 个。

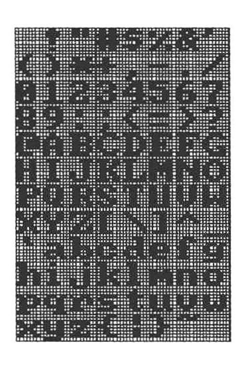
与水平方向类似，垂直方向上 525 个像素也不是都可用。原因是，像素在屏幕的顶部、底部以及垂直回归期间都会有所损失。当计算机采用电视机作为显示器时，就不依赖于隔行扫描技术了，垂直方向上有着合理的像素数目 200。

因此我们可以说，早期普通电视机上的视频适配器的分辨率为 320×200，即水平方向上有 320 个像素、垂直方向上有 200 个像素。



如何确定上面网格中像素的总数呢？你可以去数一下，也可以简单地用 320 乘以 200 得出结果 64,000。每个像素可以是黑色或白色，或者为某一特定的颜色，这要取决于视频适配器的配置。

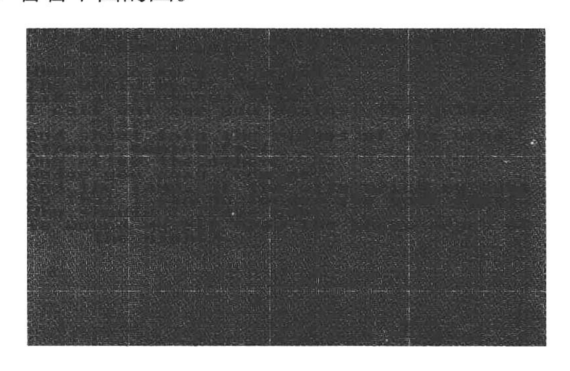
现在我们想在显示器上显示出一些文本字符，那么到底能显示出多少呢？很明显，这取决于每个字符所用的像素数。下面是一种可行的方法，每个字符使用 8×8 的网格（64 个像素）。



上图中显示出的字符对应的 ASCII 码在 20h～7Fh 区间（在 00h～1Fh 的 ASCII 码字符都是无法显示的字符）。

每个字符都被定义为一个 7 位的 ASCII 码，但每个字符也与 64 比特（位）相关，这 64 比特决定了字符会显示为什么样子。当然，你也可以把这 64 位信息当做代码看待。

上面我们对字符进行了定义，使用这些定义，分辨率为 320×200 视频显示器的每一屏就能显示 25 行、每行 40 个字符，足够把艾米·洛威尔（Amy Lowell，1874-1925）的一首短诗显示出来，看看下面的图。



视频适配器中必须配置一些 RAM，用以存储所显示的内容；微处理器也必须能够向此 RAM 中写入数据以改变显示器上显示的内容。更方便的是，这个 RAM 也是微处理器存储空间的一部分。那么，上面描述的显示适配器需要多大的 RAM 呢？

这个问题并不太好回答！我们只能说，结果可能处于 1 KB ~ 192 KB 之间。

我们从最简单的情况去考虑。怎样减少显示适配器的内存需求呢？一种方法是限制适配器的功能，让其只显示文本。我们已经明确地知道，视频显示器的每屏幕能显示 25 行、每行 40 个字符，也可以说，总共能显示 1000 个字符。这样一来，视频卡上的 RAM 只需存储这 1000 个字符的 7 位 ASCII 码。1000×7bit，大小约为 1024 字节，即 1 KB。

字符生成器（character generator）也是视频适配器板上的一部分，包含了所有 ASCII 码字符的像素图，这点前面已经讲过。通常，它是只读存储器（read-only memory），即 ROM。它是一种集成电路，在生产时里面已经填入了数据，固定的地址输出的数据是不变的。ROM 中并没有数据输入信号，这点与 RAM 不同。

你可以把 ROM 看成是可以进行代码转换的电路。每片 ROM 都有 7 个地址信号（用来表示 ASCII 码）及 64 个数据输出信号，里面存储了 128 个 ASCII 码字符的 8×8 像素图。因此，ROM 可以实现 7 位 ASCII 码到 64 位码（定义了字符显示的外观）的转换。但是你有没有想过，64 个数据输出信号会使芯片变得很大。更合适的做法是，用 10 个地址信号和 8 个输出信号。其中 7 个地址信号是用来确定 ASCII 码字符的（这 7 个地址位来自视频板上 RAM 的数据输出）。其他三个地址信号则用来表示行。举个例子来说，最高行用 000 表示，最低行用 111 表示。8 个输出位就是每行的 8 个像素。

我们来做个假设，ASCII 码为 41h，就是大写的字母 A。总共有 8 行，每行 8 位。下表给出了字母 A 的 10 位地址（ASCII 码和行代码之间用空格分开）和数据输出信号。

地址	数据输出
1000001 000	00110000
1000001 001	01111000
1000001 010	11001100
1000001 011	11001100
1000001 100	11111100
1000001 101	11001100
1000001 110	11001100
1000001 111	00000000

从上表中，你能看出以 0 为背景、用 1 表示的字母 A 吗？

只显示文本的视频显示适配器还必须支持光标（Cursor）功能。光标是一个小小的下画线，用来表明从键盘上输入的下一字符会在屏幕的什么位置显示出来。光标所在的行和列常被存储在两个 8 位的寄存器中，这两个寄存器也是视频板的一部分，而且微处理器可以对其进行写操作。

有的显示适配器不仅仅只显示文本，还可以显示其他数据，我们称这样的显示适配器为图形适配器（图形显卡）。通过向图形显卡上的 RAM 写入数据，微处理器就可以画出图形了，当然能显示各种大小和样式的文本。相比较而言，图形显卡要比只显示文本的显卡所需的存储空间更大。320×200 的图形显卡有 64,000 个像素，如果每个像素需要 1 位 RAM，那么这样的图形显卡就需要 64,000 位的 RAM，即 8000 字节。然而，这只是最低的要求。1 位是和 1 个像素相对应的，只能用来表示两种颜色——例如黑白两色。0 可能对应于黑色像素，1 可能对应于白色像素。

让我们仔细观察一下黑白电视机，很快会发现，它们不仅仅只显示黑色和白色，还能显示不同灰度的色彩。为了让图形显卡拥有这种功能，通常每个像素对应于 RAM 中的一整个字节，其中 00h 表示的是黑色，FFh 表示的是白色，介于两者之间的数值对应不同的灰度。一个 320×200 的视频板若能显示 256 种灰度，就需要 64,000 字节的 RAM。这与一直在讨论的某个 8 位微处理器的整个地址空间非常接近。

如果想显示出丰富多彩的颜色，每个像素就需要至少 3 个字节。如果现在你手头有放大镜的话，不妨用它观察一下彩色电视机或计算机视频显示器，你会发现，每种颜色都是由红、绿、蓝三原色的不同组合而形成的。为了获取所有的颜色，三原色中每种颜色的强度都需要用一个字节来表示。这么算来，就需要 192,000 字节的 RAM（更多有关彩色图形的内容将在本书最后一章介绍）。

图形显卡到底能显示出多少种不同的颜色呢？这与每个像素所赋予的比特数是有关的。对于这种关系，你可能会感到很熟悉，因为本书中讲到的很多编码都与之类似，它们都涉及 2 的幂，它们之间的关系如下：

$$\text{颜色数量} = 2^{\text{每个像素赋予的比特数}}$$

在标准的电视机上，320×200 的分辨率是所能达到的最高分辨率。正是由于这样的原

因，我们要为计算机特制显示器，以使其具有比电视机更高的带宽。1981 年，第一台显示器随 IBM PC 一起销售，它可以显示 25 行，每行 80 个字符。这正是使用在 IBM 大型机上的 CRT 显示器能显示的字符数目。对于 IBM 来说，80 个字符具有特殊的意义，为什么这样说呢？因为它和 IBM 的打孔卡片（punch card）上的字符数目一样。的确，早期连接到主机上的 CRT 显示器常被用来显示打孔卡片上的内容。偶尔，你会听到有人称只显示字符的视频显示器为卡片，当然这是一种过时的叫法。

这么多年以来，视频适配器的分辨率以及能显示的颜色不断增加，这两者也成为了视频显示适配器的重要参数。到了 1987 年，水平 640 像素、垂直 480 像素的视频适配器被 IBM 的 PS/2 个人计算机和苹果公司的 Macintosh II 机采用，这种适配器的出现起到了里程碑的作用，因为从那时起 640×480 就是视频分辨率的最低标准了。

640×480 的分辨率具有很重要的意义。也许你可能无法相信，它之所以那么重要，是因为它和托马斯·爱迪生（Thomas Edison，1847-1931）有关。大概在 1889 年，爱迪生及他的工程师威廉·肯尼迪·劳里·迪克生（William Kennedy Laurie Dickson）正在进行活动电影摄影机和活动电影放映机的研究，他们决定：让电影图像的宽比高多出 1/3。图像的宽和高之比，称为屏幕长宽比（aspect ratio）。通常，我们把爱迪生和迪克生所确定的这个比表示成 1.33:1，或者不想使用小数点的话，就表示成 4:3。60 多年了，大多数电影一直采用这个比例，电视机也是如此。但在 20 世纪 50 年代早期，好莱坞引入宽屏（widescreen）技术，与电视展开竞争，并最终打破了这个比例。

多数计算机的显示器的长宽比也是 4:3，如果你不信的话，可以用尺子实际量一下，就可以证明我所说非虚。640×480 分辨率也是这个比例。这就说明（打个比方）100 个像素的水平线和 100 个像素的垂直线有着相同的物理长度。对于计算机图形学来说，这是个非常重要的特性，我们称为正方形像素（square pixel）。

如今，视频适配器和显示器都支持 640×480 的分辨率，但同样也支持多种其他的视频模式，包括 800×600、1024×768、1280×960、1600×1200。

我们常常认为计算机的显示器和键盘之间存在某种联系——在键盘上输入什么，显示器就会显示什么——但物理上它们是分开的。

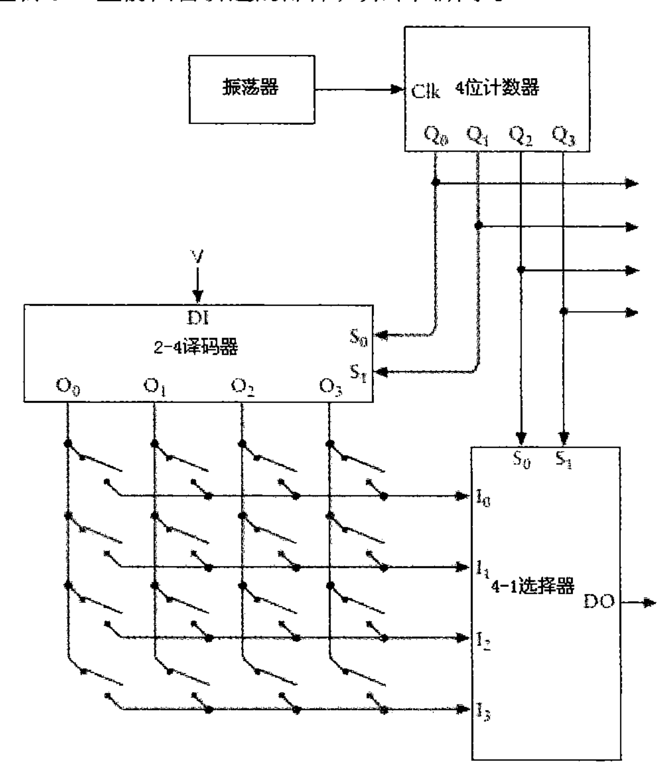
其实键盘上的每个按键就是一个简单的开关。按键按下，开关就会闭合。现在个人

计算机的键盘有 100 多个按键，但类似于打字机的键盘可能只有 48 个按键。

如果要让连接到计算机上的键盘能正常工作的话，就需要配备一些硬件来为每个按键提供唯一的代码，以便区分哪一个按键被按下了。假定这个代码就是按键的 ASCII 码，这样可行吗？你的答案或许是肯定的，但要设计出能识别 ASCII 码的硬件却是不切实际的。举例来说，键盘上的按键 A 对应的 ASCII 码可能是 41h，也可能是 61h，具体是哪个，还取决于用户是否按下了 Shift 键；另外，现在计算机键盘上有很多的按键并没有 ASCII 码与之对应。我们称键盘硬件提供的代码为扫描码（scan code）。当按下键盘上的某个按键时，一小段计算机程序就会计算出这个按键对应的 ASCII 码（如果有的话）。

这里为了避免键盘硬件的电路图太复杂，假设键盘上只有 16 个按键。任何一个按键被按下，键盘硬件就会产生一个 4 位的代码，二进制数值范围是 0000 到 1111。

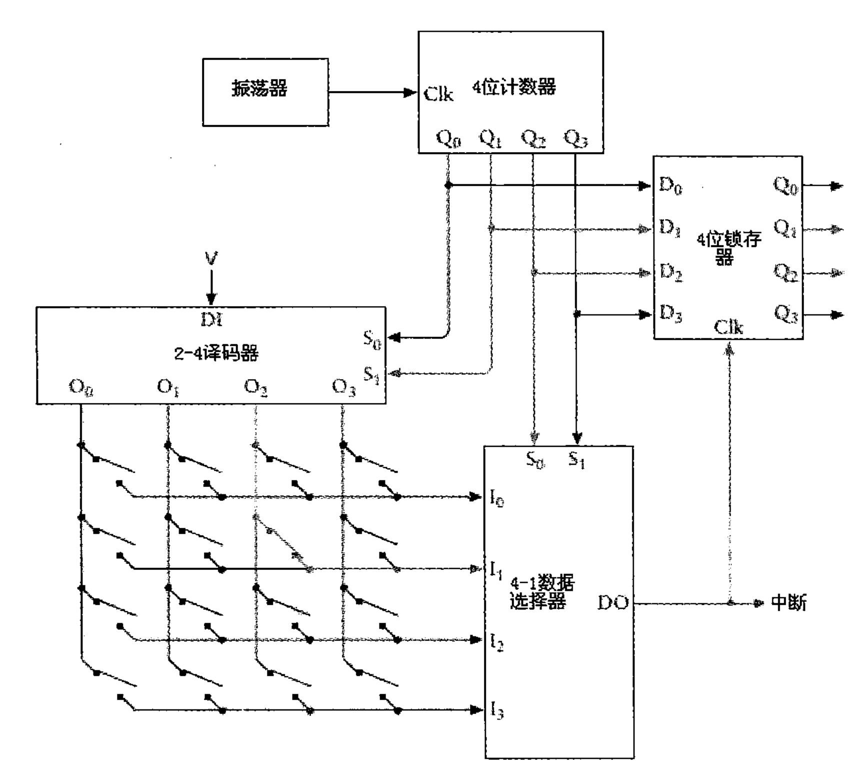
键盘硬件包含了一些前面曾讲过的部件，如下图所示。



上图左下部分所示的是键盘的 16 个按键，简单地用开关表示。4 位的计数器在按键

对应的 16 个编码间快速且重复地循环着，循环的速度必须足够快，以保证在按下并松开一个按键之前循环已经结束。

4 位计数器的输出同样也是 2-4 译码器和 4-1 数据选择器的输入。在没有按键按下的情况下，选择器的输入全都不为 1，因此，其输出也不为 1；一旦有某个按键被按下，而且与 4 位计数器某一特定输出相对应，那么选择器的输出就为 1。例如，如果右上角对角线方向的第二个开关被按下，且计数器的输出是 0110，选择器就会输出 1，如下图所示。



0110 就是这个按键的代码。在这个按键按下的情况下，计数器的其他输出都不会使选择器的输出为 1，也就是说每个按键都代码都是唯一的。

扫描码的位数是由键盘上按键的数目确定的。如果键盘上有 64 个键，就需要 6 位的扫描码，也就需要一个 6 位的计数器。用一个 3-8 译码器和一个 8-1 选择器就可以把这些按键组成一个 8×8 的阵列。如果键盘上的按键数目为 65～128 个，就需要 7 位的扫描码。你就可以用一个 4-16 译码器和一个 8-1 的选择器（或者一个 3-8 译码器和一个 16-1 选择

器）把这些按键组成一个 8×16 的阵列。

在这个电路中，接下来将会发生什么事情呢？这取决于键盘接口。每个按键都应该在 RAM 中拥有 1 位的存储空间，这是设计键盘硬件时该考虑的事情。而且这些 RAM 是由计数器寻址的，RAM 的内容为 0 或 1，具体是什么值取决于按键按下（RAM 为 1）与否（RAM 为 0）。微处理器是可以读取 RAM 中的内容的，并通过内容判断每个按键的状态。

中断信号是键盘接口一个很有用的信号。回想一下前面讲过的内容，我们知道 8080 有一个输入信号允许外部设备中断当前微处理器正进行的工作。微处理器是通过从内存中读取一条指令来响应中断的，通常是一条 RST 指令。这条指令使微处理器跳转到内存中一个特定的区域并执行其中的中断处理程序。

最后，我们要介绍一下能够长期存储信息的外围设备。前面曾提到，无论是用继电器、电子管，还是用晶体管作为介质构成随机访问存储器，一旦掉电，它存储的内容就会丢失。正因如此，能够在掉电时长期保存信息的存储器，是一台完整的计算机不可或缺的组成部分。长期以来，人们通过在纸上或卡片上打孔来保存永久信息，IBM 打孔卡片是其中典型的代表。在早期小型计算机中，为了能够长久保存程序和数据，通常在滚动的纸带上打孔，而在需要时，这些程序和数据可以从纸带加载到内存。

打孔卡片和纸带的使用也不是尽善尽美的，它也存在一些问题。首先是介质的不可重用性，一旦打孔卡片或纸带被打孔后就很难还原为原来的状态。其次是效率很低，假如你有机会看到当时纸带上保留的某一比特信息，就会发现这种做法实在太浪费纸带了。

正是由于这些原因，打孔卡片和纸带慢慢退出了历史的舞台。磁介质存储器（magnetic storage）逐渐发展成目前最为流行的长期存储器。磁介质存储器的起源要追溯到 1878 年，这一年美国工程师奥柏林·史密斯（Oberlin Smith，1840-1926）描述了它的工作原理。1898 年，即工作原理被提出 20 年后，第一块可用的磁介质存储器问世，它由丹麦发明家巴尔德马尔·波尔森（Valdemar Poulsen，1869-1942）制造。波尔森后来发明了录音电话机，当家里没人时，通过它可以记录收到的电话信息。声音通过电磁铁和可变长度的金属丝来记录，其中电磁铁是电报机里很常见的部件，它根据声音的高低来磁化金属丝。当磁化的金属丝切割电磁线圈运动的时候，产生的电流强度与其磁化程度有关。不论使用何种磁化介质，记录和读取信息都是利用电磁铁的磁头（head）来完成的。

1928年，澳大利亚发明家弗里茨·佛勒玛（Fritz Phleumer）发明了一种磁记录设备，并为其申请了专利。此设备采用在生产香烟上的金属带时所用的技术，将铁粒子覆盖在很长的纸带上。不久以后，纸带被强度更高的醋酸盐纤维素取代，而一种更耐久、更知名的记录介质也从此诞生——卷轴式磁带，它被包装在塑料盒里，可以很方便地使用。对于记录和回放音乐及视频来说，卷轴式磁带无疑是很受欢迎的介质。

1950年，雷明顿兰德公司（Remington Rand）发明了第一个用于记录计算机数字数据的商用磁带系统。当时，磁带的容量有限，一个0.5英寸的卷轴式磁带容量只有几兆字节。在早期家用计算机中，常见的盒式磁带录音机被人们用来保存信息。通过调用一些小的程序，可以将内存块中的内容保存到磁带上，以后再需要时还可以从磁带调入内存中。在第一代IBM PC上，有一个专为连接盒式磁带存储器而设计的接头。至今，磁带仍然是一种很通用的存储介质，对于那些想要长期保存的文档，磁带更是首要的选择。但是，磁盘并不是最理想的存储介质，想要快速地移动到磁盘的任一位置是不可能的，它只能顺序访问，频繁地快进和倒带会花费很多时间。

以几何学角度来看，磁盘是能够实现快速访问的介质。磁盘围绕其中心旋转，连到臂上的一个或多个磁头从磁盘外沿向中间移动，通过磁头可以快速访问磁盘上的任何区域。

在记录声音信息方面，磁盘实际上要早于磁带。早在1956年，IBM公司就发明了世界上首款用来存储计算机数据的磁盘驱动器，称为RAMAC（Random Access Method of Accounting and Control，计算与控制过程中的随机访问模式），它由50个金属盘片组成，直径2英尺，存储量为5 MB。

自从磁盘被用作记录信息以来，它的体积越来越小而容量越来越大，习惯上将磁盘分为软盘（floppy disk）和硬盘（hard disk，或fixed disk）。软盘是由单面覆盖磁性物质的塑料片组成，外面由厚纸板或塑料包装，起到保护作用（塑料包装主要是防止磁盘弯折，虽然现在的磁盘看起来不像以前的软盘那么松软，而且还有很多的区别，但软盘这个名字一直在用，延续至今）。在使用软盘的时候，必须将其插入到软盘驱动器，软盘驱动器是一个连接到计算机的部件，通过它可以读/写磁盘中的内容。早期的软盘直径为8英寸，第一代IBM PC使用的是5.25英寸的软盘，不过，现在最流行的是3.5英寸的软盘。软盘有很大的灵活性，可以实现在不同计算机之间传递数据。对商用软件来说，磁盘仍然是

一个不可或缺的发行媒介。

硬盘是由多个金属磁盘构成的，它永久驻留在驱动器里。相对于软盘来说，它的存取速度更快、存储量更大，唯一的缺点是硬盘本身是固定的，不能移动。

磁盘的表面被划分成许多同心圆，称为磁道（tracks），每个磁道又被划分成像圆饼切片形状的扇区（sectors），每个扇区可以存放一定数量的字节，通常为 512 字节。第一代 IBM PC 上使用的软盘只有一面可以存储信息，直径 5.25 英寸，被划分成 40 个磁道，每个磁道 8 个扇区，每个扇区存储 512 字节数据。通过计算，每个软盘可以存储 163,840 字节数据，即 160 KB。现今 PC 兼容机使用的软盘通常有两面，每面 80 个磁道，每个磁道 18 个扇区，每个扇区 512 字节，总的磁盘容量是 1,474,560 字节，即 1440 KB。

1983 年，IBM 在其 PC/XT 上率先使用了硬盘驱动器，当时硬盘的容量仅仅只有 10 MB。自此之后的 16 年里，硬盘的容量扩大了上百倍，而价格一直在下降。1999 年，20 GB（200 亿字节）容量的硬盘驱动器诞生了，而售价却不到 400 美元。

软盘和硬盘通常有它们自己的电气接口，除此之外，为了能和微处理器交互数据，这些电气接口与微处理器之间还需要有额外的接口与之相连。现在流行的硬盘驱动器标准接口有：小型计算机系统接口（Small Computer System Interface，SCSI）；增强的小型设备接口（Enhanced Small Device Interface，ESDI）和集成设备电气接口（Integrated Device Electronics，IDE）。这些接口都利用直接内存访问（direct memory access，DMA）技术来使用总线，DMA 可以不经过微处理器，实现数据在随机访问存储器和硬盘之间直接传送。这样的传送是以块为单位进行的，每次传输的块大小是磁盘扇区字节数的倍数，通常是 512 字节。

由于经常听到关于兆字节和吉字节之类的相关术语方面的谈论，让很多家用计算机的初学者感到困惑：到底随机访问存储器和磁盘存储器有什么区别？不过最近几年推出了一条分类规则，这让人们对术语的困惑减少了许多。这条规则规定：memory（内存）仅仅表示半导体随机访问存储器；storage（存储器）用来指任何的存储设备，通常包括软盘、硬盘和磁带。在本书中，尽量遵循这些规则，它的确能够给我们带来许多好处。

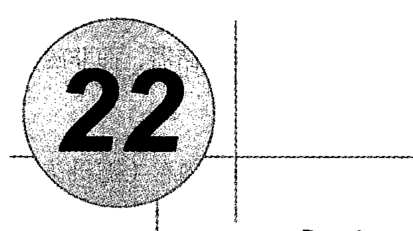
实际上，存储的信息是否易失，是随机访问存储器与磁介质存储器的主要区别。随机访问存储器是易失性存储设备，一旦掉电，存储内容将会消失；而磁介质存储器是永

久性存储设备，像软盘和磁盘，除非故意删除或写覆盖，否则数据将会一直保留不变。如果对微处理器的工作原理很了解，就会观察到随机访问存储器和磁介质存储器之间的显著区别，例如当微处理器发出一个地址信号，通常是寻址随机访问存储器，而非磁介质存储器。

微处理器不能直接从磁盘读取数据，需要将所需的数据从磁盘调入内存（随机访问存储器），然后它才能对其访问，当然这需要额外的步骤。微处理器还需要执行一段小程序，这段程序会访问磁盘，并将数据从磁盘调入内存。

关于随机访问存储器和磁介质存储器之间的差别，有个形象的比喻可以帮助我们加深理解：随机访问存储器就像办公桌的桌面，上面的任何东西都可以拿来直接使用；而磁介质存储器就像一个文件柜，里面的东西不能直接使用，如果想要使用放在文件柜里的某件东西，你需要站起来，走到文件柜前，查找需要的文件，然后带回桌面。如果桌面太拥挤，没有空间放置需要的文件，还需要把桌面上暂时不用的东西先放回到文件柜中。

这个比喻恰到好处，实际上，存储在磁盘上的数据的确是以文件（files）作为实体来存放的。存储和检索文件是操作系统（Operating System）很重要的一个功能，关于操作系统的相关知识，下一章将会进行专门介绍。



操作系统

一直以来，有一种想法在我们脑子里涌动：亲手去组装——在想象中进行虚拟组装也可以——一台近似完整的计算机。一块微处理器、一些随机访问存储器、一款键盘、一台视频显示器和一个磁盘驱动器是这台计算机所拥有的部件。当所有的硬件各就各位，我们激动地盯着计算机的开关，伸出手来给它上电，将这台计算机从沉睡中唤醒。或许上面描述的这一切会在你的脑海中产生一副维克多·弗兰肯斯坦[1]（Victor Frankenstein）组装怪物时的场景，甚至你或许还会想起老木偶匠盖比特（Geppetto）正在雕刻木偶匹诺曹（Pinocchio）的情形。

但我们还是漏了一些东西，既不是雷霆万钧的威力，也不是对着流星许下的纯洁愿望。让我们继续投入到工作中：运行这台新组装的计算机，请告诉我你眼前出现了什么?

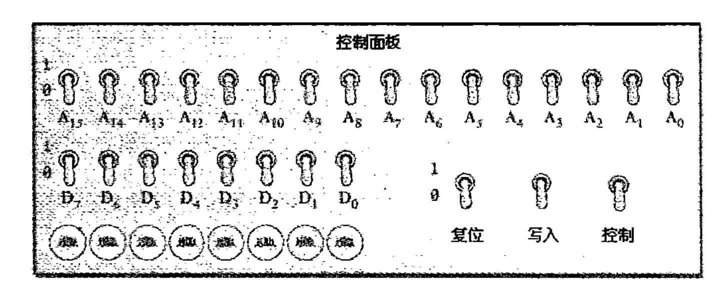
当阴极射线管加热之后，一串排列整齐——而又完全随机的——ASCII 码字符阵列出现在屏幕上。不出我们所料，掉电的时候，半导体存储器中的内容就会被全部清零；而

1 “弗兰肯斯坦”是英国著名小说家玛丽·雪莱（Mary Shelley）创作的同名小说中一个疯狂科学家的名字。小说中，弗兰肯斯坦用许多碎尸块拼接成一个“人”，并用闪电将其激活。玛丽·雪莱是英国著名浪漫主义诗人雪莱的妻子，被誉为科幻小说之母。

首次给它上电的时候，它将处于随机且不可预测的状态。同样，我们用来构建微处理器的所有 RAM 都包含随机的字节。如果可能，开机后微处理器会将这些随机字节解释为机器代码并执行。不用担心会因此发生什么糟糕的事情——计算机不会因此而坏掉——但是，计算机也无法完成任何有意义的工作。

这里我们漏掉的就是软件。当一个微处理器首次上电或复位时，它会从特定的内存地址开始执行机器代码。在英特尔的 8080 系统中，这个地址就是 0000h。在一台设计精良的计算机中，通过上电启动，将会有一条机器代码指令被载入到该内存地址中（一般情况下是一段程序的第一条指令）。

机器代码指令是怎么加载到那个内存地址中的呢？把软件安装到一台新设计的计算机的过程，可能是整个过程中最令人困惑的部分了，怎样理解它呢？让我们先从第 16 章所讲的一个控制面板入手吧，这个控制面板的功能是把字节写入随机访问存储器，之后还可将其读出。



与前面介绍过的有所不同，这个控制面板上设计了一个复位开关。复位开关与微处理器的复位输入相连接。一旦复位开关闭合（置 1），微处理器就会停止工作；当此开关断开后（置 0），微处理器就开始执行机器代码。

使用此控制面板的方法是：打开复位开关，微处理器复位并停止执行机器代码；打开控制开关，就会接收总线上的地址信号和数据信号。在该状态下，你可以通过开关 A_0 ~ A_{15} 来指定一个 16 位的存储器地址；通过灯泡 D_0 ~ D_7 的明灭组合来显示该存储器地址中的 8 位数据。那么怎样把一个新的字节写入到此地址中呢？首先通过开关 D_0 ~ D_7 来设置想要写入的字节，然后把写入开关先打开再关闭。当你已经完成向存储器中插入字节的工作后，关上控制及复位开关，微处理器就会执行程序。

上面这个过程展示了向这台我们刚刚打造出来的计算机输入第一条机器代码的步骤。

不言而喻，这是一个耗时耗力的过程。在这个过程中一些小错误是在所难免的。看看你的手指，或许已经磨出了水泡，你的大脑也感觉一片混乱，而这一切都是工作的代价。

但当你开始用视频显示器显示程序运行的结果时，到底是什么使这一切都变得简单、方便呢？上一章中我们讲到只显示字符的视频显示器，它有一个 1 KB 的随机访问存储器，可以存储 25 行、每行 40 个字符的 ASCII 码。程序将要显示的内容写入到此存储器中，其方法与向计算机中其他存储器中写入数据的方法一样。

尽管把程序的输出结果显示在视频显示器上看似简单，实则不然。例如，你编写了一个程序用来完成某个计算任务，如果计算结果是 4Bh，不能将这个值直接写入到视频显示器的内存中。如果犯了这样的错误，屏幕上显示的将是字母 K，因为此字母的对应 ASCII 码的值正是 4Bh。4Bh 由两个字符组成，其中 4 对应的 ASCII 码是 34h， B 对应的 ASCII 码是 42h，应该将这两个 ASCII 码写到视频显示器存储器上，才能在显示器上看到期望的数值。这里再强调一下，8 位二进制数可以表示两位十六进制数字，因此必须将每一位十六进制数字对应的 ASCII 码，写入到视频显示器的存储器中才能显示这个数。

这种转换可以通过编写小的程序来实现。下面是一段 8080 汇编程序，功能是把存储在累加器中的十六进制数（假设这个数介于 00h 与 0Fh 之间）的每一位转换成对应的 ASCII 码：

```
NibbleToAscii: CPI A, 0Ah   ; 检查是数字还是字母
               JC Number
               ADD A, 37h   ; 把 A~F 转换成 41h~46h
               RET
Number:        ADD A, 30h   ; 把数字 0~9 转换成 30h~39h
               RET
```

通过两次调用 NibbleToAscii，下面的程序实现了把累加器 A 中的一个字节转换成两个 ASCII 码对应的数字，分别存放在寄存器 B 和 C 中。

```
ByteToAscii:   PUSH PSW              ; 保存累加器 A
               RRC
               RRC
               RRC
               RRC                   ; 获取高半字节
               CALL NibbleToAscii    ; 转换成 ASCII 码
               MOV B, A              ; 把结果存入寄存器 B 中
               POP PSW               ; 取出原始的 A
               AND A, 0Fh            ; 获取低半字节
               CALL NibbleToAscii    ; 转换成 ASCII 码
```

```
MOV C, A                    ; 把结果存入寄存器 C 中
RET
```

通过这些程序，可以把一个用十六进制表示的字节显示在视频显示器上。进一步来讲，如果想把它转换成十进制数，还需要做些别的工作。这个过程与把一个数从十六进制转换成十进制的过程（用 10 除几次）十分相似。

记住，到此为止实际上你还没有将汇编语言程序写入到内存。你需要把汇编语言写到纸上，将它们转换成机器代码后才写入到内存中。直到第 24 章我们一直都会采用这种“手工汇编”的方式。

控制面板的确不需要很多硬件支持，但它不便于使用，因为它有着最糟糕的输入/输出形式。我们甚至可以从头开始独立建造一台计算机，但仍然无法改变这样糟糕的输入/输出方式——通过按键输入 0 和 1——这的确让人尴尬。如何把控制面板去掉是要解决的首要问题。

实现按键来控制输入/输出的不二之选就是键盘。前面我们已经搭建了一个计算机键盘，每次有按键按下的时候，就会产生一个中断信号送至微处理器。计算机内有中断控制芯片，通过执行一条 RST 指令使得微处理器响应这次中断，例如 RST 1，微处理器执行这条指令，把当前程序计数器的值压入到堆栈中，然后跳转到地址 0008h 处。可以直接在这片地址空间上输入一些代码（使用控制面板），这些代码称为键盘处理程序（keyboard handler）。

为了使复位后微处理器能正常工作，微处理器在复位的时候需要执行一些代码，称为初始化代码（initialization code）。堆栈指针在运行初始化代码的时候会被设置，以保证堆栈处在内存的有效区域内。为了不让屏幕上显示随机字符，初始化代码还把视频显示器内存中的每个字节设置成十六进制数 20h，在 ASCII 码中这是一个空格符。此外，初始化代码还要把光标定位在第一行第一列的位置——OUT（Output）指令可以完成这一操作：光标在视频显示器上是以下画线的形式出现的——它可以显示出下一个要输入字符的位置。为了使微处理器能响应键盘中断，必须设置 EI 指令开中断，而 HLT 指令可以使微处理器停止工作。

上面讲述的就是初始化代码的作用。执行了 HLT 指令后，计算机则处于停机状态。为了把计算机从停机状态唤醒，只能通过控制面板的复位信号或者键盘的中断信号来实现。

键盘处理程序的规模要远大于初始化代码，这个程序才是真正响应键盘事件的代码段。

任何时候，只要键盘上的一个按键被按下，微处理器就会响应本次中断，并从初始化代码末尾的 HLT 语句跳转到键盘处理程序。键盘处理程序利用 IN（Input）指令用来检查是哪个按键被按下，并根据这个按键执行相关的操作（就是说，键盘处理程序对每个按键进行相应的处理），然后执行 RET（Return）指令以返回 HLT 语句，等待另一个键盘中断。

当你按下字母、数字或者是标点符号键的时候，键盘扫描程序就会启用键盘扫描码，并根据 Shift 键是按下与否确定相应的 ASCII 码。接下来我们要做的，就是要把这个 ASCII 码写到视频显示器的内存中，当然这不是随意的，而是要写在光标所在的位置。这样的一个过程，我们可以很形象地称之为按键到显示器的回显（echoing）。光标会随着字符的写入而移动，换言之，它总会出现在刚显示的字符后面的空格处。通过键盘，可以输入一串字符，然后把它们在屏幕上显示出来。

当按下回退键（相应的 ASCII 码值是 08h）时，最后写入视频显示器内存中的字符会被键盘处理程序删除（其实并不复杂——我们只要把空格符对应的 ASCII 码——20h 写入到那个内存位置处就行了）。在这个过程中，光标会移回一格。

通常我们在输入一行字符时，错误是难免的，这时就需要用退格键来改正错误，然后按下回车键。回车键并不难找到，键盘上标有“Enter”字样的按键就是。打字员在电动打字机上按下“Return”键表示已经完成一行文字的输入，同时也表明他们已经做好了输入下一行的准备，光标会指向下一行的开始。同样，在计算机中“Enter”键用来实现相同的功能，结束一行的输入并转到下一行。

当键盘处理程序对“Return”或“Enter”键（对应的 ASCII 码是 0Dh）进行处理时，它把视频显示器内存中的这一行文本解释为计算机的一条命令（command），换言之，键盘处理程序的任务是执行此命令。实际上，在键盘处理程序内含有一个命令处理程序（command processor），它可以解释如下三条命令：W 命令、D 命令和 R 命令。下面我们来深入地理解一下它们。

首先是 W 命令。它是以 W 开头的文本行，此命令用来把若干字节写入（Write）到内存中。例如输入到屏幕上的一行内容如下所示：

```
W 1020 35 4F 78 23 9B AC 67
```

运行这条命令，命令处理程序会从内存地址 1020h 处开始，把 35、4F 等十六进制表示的字节写入内存中。要完成这项工作，键盘处理程序需要把 ASCII 码转换成字节——前面讲过把字节转换成 ASCII 码，这里其实就是它的逆变换。

接下来是 D 命令。它是以 D 开头的文本行，此命令用来把内存中的一些字节显示（Display）出来。例如输入到屏幕上的一行内容如下所示：

```
D 1030
```

接收到命令后，命令处理程序会把从地址 1030h 开始的 11 个字节的内容显示出来（这里之所以说是 11 个字节，是因为在一个每行可以容纳 40 个字符的显示器上，除去显示命令与地址标识，后面能显示的也只有这么多了）。有了这条命令，就可以很方便地查看内存中的内容了。

最后是 R 命令。它是以 R 开头的文本行，表示运行（Run）。该命令的形式如下：

```
R 1000
```

执行此命令意味着“处理器会运行从地址 1000h 开始的一段程序”。首先命令处理程序把 1000h 存储在寄存器对 HL 中，接着执行指令 PCHL，这条指令的功能是，把 HL 所存储的值加载到程序计数器中，然后跳转到程序计数器指向的地址并运行程序。

键盘处理程序及命令处理程序简化了很多工作，可以说是计算机发展的一个里程碑。一旦使用了它，就无须理会那个令人难以忍受的控制面板了。我们不得不承认，使用键盘输入更简单、更快、更好，这是其他方法无法媲美的。

但是，你仍然没有摆脱之前的老问题，一旦关掉电源，你辛辛苦苦所输入的数据会全部消失。当然，你可以把所有新代码存到只读存储器（ROM）中。还记得上一章中，我们讲到的那个 ROM 芯片吗？它就包含了把 ASCII 码字符显示到视频显示器上所需的全部点阵模式。这里假定，这些数据在厂家制造芯片时已经配置好了，当然你也可以对 ROM 芯片进行编程。可编程只读存储器（Programmable Read-Only Memory，PROM）只能编程一次；而可擦除可编程只读存储器（Erasable Programmable Read-Only Memory，EPROM）可重复擦除和写入，该芯片的正面开有一个玻璃窗口，让紫外线透过这个孔照射内部芯片就可以擦除其中的内容了。

回忆前面曾讲过的内容，你一定会想起，RAM 板与一个 DIP 开关相连，有了这个开

关就可以设定 RAM 板的起始地址了。8080 系统在初始化时，其中一个 RAM 板的起始地址被设置为 0000h。但是如果有 ROM 的话，这个地址就会被其占用，而 RAM 板则转到更高的地址。

命令处理程序的使用极大地推动了计算机的发展，通过它不仅可以更快地向存储器输入数据，更重要的是，计算机变得可交互（interactive）了。当你通过键盘输入一些内容，计算机会立即做出响应，把你所输入的内容显示出来。

将命令处理程序存储到 ROM 后，就可以执行操作了：把内存中的数据写入到磁盘驱动器（可能是按照与磁盘的扇区大小一致的块的形式），然后再把数据读回到内存中。因为掉电的时候，RAM 中的内容会丢失，所以与 RAM 相比，把程序和数据存储到磁盘中会更安全（它们不会因为突然断电而丢失数据），就灵活性来说，也比存储到 ROM 中要好。

仅仅有上面的命令还是不够的，还需要向命令处理程序中添加新命令。例如，S 命令表示存储（Store）：

```
S  2080  2  15  3
```

运行这条命令后，在磁盘的第 2 面、第 15 道、第 3 扇区中将存放起始地址为 2080h 的内存块数据（被存放内存块的大小是由磁盘扇区的大小决定的）。类似地，还可以通过加载（Load）命令，把磁盘上相应扇区的内容写回到内存中，如下所示：

```
L  2080  2  15  3
```

当然，还要把存储的位置记录下来，这是必须要做的。你可以用手头上的纸和笔来完成。需要注意的是：你不能把位于某个地址处的代码又加载到内存的另外一个地址中，如果这样的话，程序将不能正常运行。具体来说，程序代码在内存中改变位置后，其跳转（Jump）和调用（Call）指令标识的依然是原来的地址，所以运行时会报错。也存在这样的情况，程序比磁盘的扇区大，这时就需要多个扇区来存放程序。而磁盘上某些扇区已被其他程序或数据占用了，而另外一些扇区是空闲的，可能在磁盘上找不到一块足够大的、连续的扇区来存放程序。

最后，所有的东西存储在磁盘的什么位置，都需要你手工地记录下来，这个工作挺多，也挺麻烦。出于这个原因，文件系统（file system）应运而生。

文件系统是磁盘存储的一种方法，就是把数据组织成文件（file）。简单地说，文件是

相关数据的集合，占用磁盘上一个或多个扇区。更重要的是，你可以为每个文件命名，这有助于记下文件里存放的内容。想象一下，磁盘是不是类似于一个文件柜，每个文件都有个小标签，标签上有文件的名称。

操作系统（operating system）是许多软件构成的庞大程序集合，文件系统就是其中的一部分。前面讲到的键盘处理程序和命令处理程序最终也能经过拓展，演变成为操作系统。那么操作系统到底能够做些什么、又是如何工作的呢？这里撇开操作系统漫长的发展演化过程，把重点放在刚才提出的问题上，目的就是试图让大家了解操作系统的工作机制。

如果你对操作系统的发展史有一定了解的话，你就会知道 CP/M（Control Program for Micros）是最重要的 8 位微处理器操作系统，它是 20 世纪 70 年代中期由加里·基尔代尔（Gary Kildall，生于 1942 年）专门为 Intel 8080 微处理器而开发的，加里后来成了 DRI（Digital Research Incorporated）公司的创始人。

CP/M 操作系统是存放在磁盘上的。单面、8 英寸的磁盘是早期的 CP/M 最常用的存储介质，它有 77 个磁道，每个磁道有 26 个扇区，每个扇区的大小是 128 个字节（总共算下来共有 256,256 个字节），CP/M 系统存放在磁盘最开始的两个磁道。在启动计算机时，需要把 CP/M 从磁盘调入到计算机的内存中，下面将介绍这一过程是如何进行的。

上面我们讲过，存放 CP/M 本身只占用 2 个磁道，那么剩下的 75 个磁道用来存储文件。CP/M 的文件系统固然简单，但两个最基本的要求还是可以满足的。首先，每个文件在磁盘中都有属于自己的名字，便于识别，这个名字也是存放在磁盘中的；实际上，文件以及读取这些文件需要的所有信息也是一起存储在磁盘中的。其次，文件存储在磁盘中不一定要占据连续的扇区空间，可以想象，由于大小不同的文件会被经常地创建和删除，磁盘上的可用空间就会很零碎。那么如何将文件存储在零碎的空间里呢？这主要得益于文件系统具有很强大的管理功能，它可以把一个大文件分散存储在不连续的磁盘上。

上面曾提到过剩下的 75 个磁道中的扇区用来存放文件，这些扇区按分配块（allocation blocks）进行分组。每个分配块中有 8 个扇区，总计 1024 个字节（每个扇区大小是 128 字节）。可以计算，在磁盘上共有 243 个分配块，编号为 0～242。

目录（directory）区占用最开始的两个分配块（编号为 0 和 1，总共 2048 字节）中。

目录区是磁盘中的一个非常重要的区域，磁盘文件中每个文件的名字和其他一些重要信息都存储在该区域，根据目录就能够很方便、高效地查找文件。存放目录也需要占用空间，磁盘上每个文件对应的目录项（directory entry）大小均为 32 字节，由于目录区大小为 2048 字节，所以这个磁盘上最多可以存放 64（2048/32）个文件。

为了能够根据目录找到相应的文件，每一个 32 字节的目录项包含以下信息。

字　节	含　义
0	通常设为 0
1 ~ 8	文件名
9 ~ 11	文件类型
12	文件扩展
13 ~ 14	保留（设置为 0）
15	最后一块扇区数
16 ~ 31	磁盘存储表

在目录项中，第一个字节用来设置文件的共享属性，只有文件系统被两个或更多人同时共享时才设置此字节为 1。在 CP/M 中，这个字节跟第 13、14 字节一样，通常设置为 0。

CP/M 中每个文件的文件名由两部分构成，第一部分称为文件名（filename），文件名最多由 8 个字符构成，目录项的第 1 ~ 8 字节用来存储文件名。第二部分称为文件类型（file type），最多由 3 个字符表示，这些字符存储在目录项的第 9 ~ 11 字节中。我们会经常遇到一些常见的标准文件类型，例如：TXT 表示文本文件（此文件只包含 ASCII 码）；COM（command 的简写）表示这个文件中存放的是 8080 机器码指令，或者说是一段程序。当命名一个文件时，通过一个点来隔开这两部分，如下所示：

MYLETTER.TXT

CALC.COM

人们习惯形象地称这种文件命名方式为 8.3，就是说在点号隔开的前半部分最多有 8 个字母，后半部分最多有 3 个字母。

接下来介绍一下如何利用目录项查找文件。目录项中的第 16 ~ 31 字节是磁盘存储表，它能够标明文件所存放的分配块。假设磁盘存储表的前 4 项分别为 14h、15h、07h、23h，

其余项全为 0，这表明文件占用了 4 个分配块的空间，大小为 4 KB，而实际上文件可能并没有用完 4 KB 空间，因为最后一个分配块往往只有部分扇区被使用。目录项的第 15 字节表明最后一个分配块到底用到了多少个 128 字节的扇区。

磁盘存储表的长度为 16 字节，最多可以容纳 16,384 字节（16 KB）的文件，如果文件的长度超过 16 KB，就需要用多个目录项来表示，这种方法称为扩展（extents）。如果一个大文件使用目录项扩展来，则将第一个目录项的第 12 字节设置为 0，第二个目录项的第 12 字节则设置为 1，依此类推。

文本文件对我们并不陌生，通常也称之为 ASCII 文件（ASCII file）、纯文本文件（text-only file）或纯 ASCII 文件（pure-ASCII file），当然还有其他一些类似的名称。ASCII 码（包括换行符和回车符）是文本文件中唯一包含的字符，文本文件通俗易懂，便于浏览。除了文本文件外，其余的文件称为二进制文件（binary file），在 CM/P 中，COM 文件存放的是二进制的 8080 机器码，它是二进制文件。

假设一个小文件中包含三个 16 位数，如：5A48h、78BFh 和 F510h。如果此文件是二进制文件，只要 6 字节就可以了。

```
48  5A  BF  78  10  F5
```

这是采用 Intel 格式来存储的，放在前面的是低位，放在后面的是高位。并不是所有的数据都是按照这种格式来存储的，例如为 Motorola 处理器编写的程序更倾向于按以下的方式来组织文件：

```
5A  48  78  BF  F5  10
```

假如上面的三个 16 位数用 ASCII 文本文件来存放，则文件中保存的数据如下所示：

```
35 41 34 38 68 0D 0A 37 38 42 46 68 0D 0A 46 35 31 30 68 0D 0A
```

上面的这些字节是数字和字母的 ASCII 码表示形式，用回车符（0Dh）和换行符（0Ah）来表示每一个数的结束。因为文本文件可以不通过解释相应的 ASCII 字节串来显示字符，而是将字符本身直接显示，所以显得更加方便，如下所示：

```
5A48h
78BFh
F510h
```

也可以用如下的形式来表示包含这三个数的 ASCII 文本文件：

```
32 33 31 31 32 0D 0A 33 30 39 3 31 0D 0A 36 32 37 33 36 0D 0A
```

这是用十进制数的 ASCII 码形式来表示上述三个数，这两种表示形式是等价的，如下所示：

```
23112
30911
62736
```

显然，文本文件更易于人们阅读，同样，与十六进制相比，十进制更符合人们的习惯，没理由使用十六进制而拒绝十进制。

前面曾提到过，磁盘最开始的两个磁道存储 CP/M 系统本身，而 CP/M 在磁盘上是无法运行的，必须将其加载到内存里。只读存储器（ROM）在 CP/M 计算机中使用得并不多，只需要用它来存放一小段称为引导程序（bootstrap loader，操作系统的其余部分可以通这段代码的自举操作被高效地引导）的代码即可。开机启动时，磁盘上最开始的 128 字节的扇区内容，会首先由引导程序加载到内存并运行，这个扇区包含有特定的代码，可以把 CP/M 中的其余部分加载到内存中，整个过程称为操作系统的引导（booting）。

操作系统的引导过程完成后，随机存储器（RAM）的最高地址区域用来存放 CP/M，加载完 CP/M 后，整个内存空间的组织结构如下所示。

地址	内存区域
0000h:	系统参数
0100h:	临时程序区域（TPA）
	控制台命令处理程序（CCP）
	基本磁盘操作系统（BDOS）
最高地址:	基本输入/输出系统（BIOS）

该图仅仅粗略地表示出了内存各构成部分，没有按比例刻画各部分所占内存的大小。控制台命令处理程序(Console Command Processor，CCP)、基本磁盘操作系统(Basic Disk Operating System，BDOS)和基本输入/输出系统(Basic Input/Output System，BIOS)是CP/M的三个组成部分，这三个部分只占用了6 KB大小的内存空间。在拥有64 KB内存空间的计算机中，大约58 KB被临时程序区(Transient Program Area，TPA)占用，但是这58 KB空间一开始时是空的。

控制台命令处理程序的功能和以前讨论过的命令处理程序是一样的。在这里，键盘和显示器组成了控制台(console)。控制台命令处理程序显示如下所示的命令提示符(prompt)：

```
A>
```

在命令提示符后面可以输入一些信息。大多数计算机可能有不止一个磁盘驱动器，第一个磁盘驱动器标为A，CP/M被装载到该驱动器中。在命令提示符后面敲入命令并按回车(Enter)键，控制台命令处理程序会执行该命令，然后将执行的结果显示到屏幕上。执行完命令后，命令提示符又会显示在屏幕上，等待下一次输入。

控制台命令处理程序只能识别一部分命令，其中最重要的命令是：

```
DIR
```

该命令用于显示磁盘的目录信息，换句话说，它列出了存储在磁盘上的所有文件。然而有时候需要查看具有特定名字和类型的文件，这时候就可以在命令中使用像“？”和“*”这样的特殊字符来限定。如果想显示当前目录下所有的文本文件，可以使用如下指令：

```
DIR  *.TXT
```

而

```
DIR  A???B.*
```

则显示所有文件名由5个字符构成，其中第一个字符是A，最后一个字符是B的文件。

如果想删除磁盘中的文件，要用到命令ERA，它是Erase的缩写，例如：

```
ERA  MYLETTER.TXT
```

用来删除名为 MYLETTER.TXT 的文件，而运行下面的命令：

```
ERA  *.TXT
```

则所有的文本文件都被删除。一旦删除文件，此文件的目录项及其所占用的磁盘空间都将被释放。

REN 也是一个常用命令，它是 Rename 的缩写，此命令可以改变文件名。如果想显示文本文件的内容，可以使用 TYPE 命令，这条命令的功能还不仅如此，要知道文本文件中仅包含 ASCII 码，所以屏幕上文件的内容也可以通过这条命令来浏览，例如：

```
TYPE  MYLETER.TXT
```

表示查看 MYLETER.TXT 文件的内容。SAVE 命令用来保存文件，它可把临时存储区域中的一个或多个 256 字节的内存块保存到磁盘中，并且给这个内存块指定一个名字。

当然，上述所介绍的都是 CP/M 可识别的命令，如果你输入一个不能被 CP/M 识别的命令，CP/M 就会默认为输入的是保存在磁盘上的一个程序名。而程序通常是以文件形式存储的，其文件类型为 COM，代表着命令。控制台命令处理程序负责在磁盘上查找此文件，如果找到，此文件会被 CP/M 从磁盘加载到临时程序区域，该区域的地址从 0100h 开始，一旦文件被调入内存即可运行。上面从流程的角度介绍了磁盘中的文件是如何被运行的，下面举个例子来说明。假如在 CP/M 命令提示符后面输入：

```
CALC
```

如果在磁盘中存在名为 CALC.COM 的文件，则该文件会被控制台命令处理程序调入到以地址 0100h 开始的内存中，接着控制台命令处理程序会转到该地址并执行这段程序。

前面介绍了如何将机器码指令插入到内存空间的任意位置并执行。但是存储在磁盘上的 CP/M 程序并不能被随意存放到内存中的任意位置，它必须被加载到指定的位置，在这里是以 0100h 开始的内存空间。

CP/M 由一些实用的程序组成，如 PIP（Peripheral Interchange Program），即外设交换程序，通过它可以复制文件。ED 是文本编辑器，可以创建和修改文本文件。在 CP/M 系统中，有很多像 PIP 和 ED 一样的程序，它们很小但可以完成简单的事务处理，这类程序称为实用（utility）程序。仅仅有这些小的简单程序是远远不够的，还有一些大的商业化

应用程序（application），比如字处理软件或计算机电子报表软件等，在 CP/M 系统中，这些软件的使用会给你的工作带来很大方便。当然，如果你是一个程序开发高手，也可以自己动手编写这些软件，这些程序的存储类型都是 COM 文件类型。

在 CP/M（跟许多操作系统一样）中，我们了解了很多内容，比如如何利用命令和实用程序对文件进行基本操作，如何将程序加载到内存中并运行等。操作系统的功能远不止如此，下面将介绍它的第三个重要功能。

前面提到过，把输出信息写到视频显示器上，或者从键盘读取输入的内容，或者读/写磁盘中的文件，这些都是运行在 CP/M 下的程序常常要做的操作。这就需要 CP/M 程序能够直接向视频显示器的内存写入输出内容，也需要 CP/M 程序能够访问键盘硬件来捕获所输入的内容，还需要 CP/M 程序能够访问磁盘驱动器来读/写磁盘扇区，然而在通常情况下程序本身是很难直接做到的。

那么有没有别的方法来实现上述的要求呢？答案是肯定的，这些常用事务由 CP/M 中的子程序集来完成，在 CP/M 下运行的程序通过调用这些子程序即可完成相应的操作。这些子程序都是专门设计的，计算机中的所有硬件都可以很容易地通过它们来访问，如视频显示器、键盘、磁盘驱动器等，而程序员无须关心这些外设实际是如何连接的。更重要的是，像磁道、扇区这类信息，程序没有必要知道，这些工作都是由 CP/M 来完成的，它可以负责读/写磁盘上的文件。

操作系统提供的第三个主要功能是让程序能够方便地访问计算机的硬件，操作系统提供的这种访问操作称为 API（Application Programming Interface），即应用程序接口。

那么如何设置和使用 API 呢？在 CP/M 下运行的程序，可以通过将寄存器 C 设置为特定的值（称为功能值），并且运行如下指令：

```
CALL 5
```

来使用 API。例如，你从键盘上按下了一个键，程序会通过执行下面的指令来获取此键对应的 ASCII 码：

```
MVI C,01h
CALL 5
```

并且将这个键的 ASCII 码值保存在累加器 A 中。类似的，运行这条命令：

```
MVI C,02h
CALL 5
```

在视频显示器上当前的光标位置将显示累加器 A 中的 ASCII 码字符，然后光标移到下一个位置。

如果程序要新建一个文件，它首先将文件名所在区域的地址保存在寄存器对 DE 中，接着执行如下代码：

```
MVI  C,16h
CALL 5
```

执行此命令，CP/M 会在磁盘上新建一个空文件。程序可以利用 CP/M 提供的其他功能来向空文件中写入内容，最后关闭（close）文件，关闭文件意味着文件使用完毕，不需要在对该文件执行任何操作了。当然，该文件可以再次被此程序和其他程序打开（open）并读取内容。

CALL 5 指令到底如何工作呢？CP/M 在内存中地址为 0005h 处设置了一条 JMP（Jump）指令，它跳转到 CP/M 基本磁盘操作系统（Basic Disk Operating System，BDOS）中的某个位置。这个区域包含许多小程序，CP/M 的每一项功能都可由它们完成。BDOS，顾名思义，它的主要功能是维护磁盘上的文件系统。文件系统经常要与终端设备打交道，所以 BDOS 经常要调用 CP/M 基本输入/输出系统（Basic Input And Output System，BIOS）中的一些子程序。这里顺便提一下，BIOS 可以对硬件进行访问，比如键盘、视频显示器和磁盘驱动器等。实际上，BIOS 是 CP/M 中唯一需要了解计算机中硬件的程序，其他一些对硬件的操作都可通过调用 BIOS 中的子程序来实现。控制台命令处理程序通过调用 BDOS 的子程序来实现自己所有的功能，而在 CP/M 中运行的程序也是这样。

对于计算机硬件来说，API 是一个与设备无关（device-independent）的接口。换言之，对于特定的机器上键盘的工作机制、视频显示器的工作机制以及磁盘扇区的读/写机制，在 CP/M 下编写的程序不需要知道也没有必要知道。程序使用 CP/M 提供的功能便可完成对键盘、视频显示器和磁盘驱动器操作，简言之，API 屏蔽了硬件之间的差异。有了 API，尽管不同计算机硬件差别很大，其访问外设的方式也不尽相同，但 CP/M 程序都可以在上面运行，从而实现了 CP/M 程序的跨平台（这里要求所有 CP/M 程序必须运行在 Intel 8080 微处理器上，或者运行在能执行 Intel 8080 指令的处理器上，例如 Intel 8085 或 Zilog Z-80

处理器）。所以，在使用 CP/M 系统的计算机中，程序对硬件访问是通过 CP/M 来实现的。如果没有标准的 API，程序必须根据不同型号的计算机进行相应的修改后，才能访问硬件。

CP/M 是 8080 中非常流行的操作系统，曾经辉煌一时，至今仍有重要的历史意义。16 位操作系统 QDOS(Quick and Dirty Operating System)的开发在很多方面都借鉴了 CP/M 的思想。QDOS 当初是专为英特尔公司的 16 位 8086 和 8088 芯片而设计的，它出自西雅图计算机产品公司（Seattle computer product）的提姆·帕特森之手。QDOS 系统被微软公司注册后更名为 86-DOS。1981 年，随着 IBM 第一代 PC 诞生，微软公司也将 86-DOS 更名为 MS-DOS（Microsoft Disk Operating System），并以此名授权给 IBM 公司用作第一代个人计算机的操作系统。这就是著名的 MS-DOS 系统，在现在的计算机中也会经常看到它的身影。虽然 16 位版本的 CP/M（称为 CP/M-86）也可用于 IBM PC，但由于 MS-DOS 的影响力更大，其很快成为了标准。其他生产 IBM PC 兼容机的厂商也被授权使用 MS-DOS（在 IBM 计算机上称为 PC-DOS）系统。

CP/M 的文件系统在 MS-DOS 没有被继续使用，在 MS-DOS 中，文件系统是以文件分配表（FAT，File Allocation Table 的简写）的形式来组织的，Microsoft 公司早在 1977 年就开始使用 FAT 了。FAT 的基本思想是：将磁盘空间分成簇（cluster）——簇的大小由磁盘空间的大小来决定——从 512 字节到 16 K 字节不等，每个文件占用若干簇。文件的目录项只记录文件起始（starting）簇的位置，而磁盘上每一簇的下一簇的位置由 FAT 来记录。

每个目录项在 MS-DOS 磁盘上占用 32 字节，其命名形式跟 CP/M 上的 8.3 形式一样，只是使用的术语有些区别：最后的三个字符称做文件扩展名，而非 CP/M 中的文件类型。MS-DOS 目录项中没有包含分配块列表，它主要包含如下所示的有用信息：文件的最后修改的日期、时间和文件的大小等。

实际上，MS-DOS 的早期版本与 CP/M 在结构上很类似，只是 IBM PC 本身在 ROM 中已经包含了一套完整的 BIOS 了，所以在 MS-DOS 中不再需要 BIOS。在 MS-DOS 中，命令处理器是一个命名为 COMMAND.COM 的文件。MS-DOS 有两种运行程序：一是以 COM 为扩展名的文件，其大小不能超过 64 KB，二是更大一些的程序，以 EXE（意思是可以被执行）为文件的扩展名，表明文件本身是可执行的。

尽管 MS-DOS 起初支持 API 函数的 CALL 5 接口，但不久 Microsoft 为新的程序重新

设计了一款新的接口。这个新的接口使用了 8086 的一个称为软件中断（software interrupt）的功能，说起软件中断，它与子程序调用很类似，只不过程序不必知道它所调用的子程序的确切地址。通过执行 INT 21h 这条指令，程序可以调用 MS-DOS 的 API 功能。

从理论上讲，应用程序并不能直接访问计算机的硬件，如果它要访问计算机的硬件，可以通过操作系统提供的接口来进行。然而实际上，在 20 世纪 70 年代末和 80 年代初，由于计算机上运行的都是小型操作系统，许多应用程序往往都绕过它们，这种情况在处理有关视频显示器任务的时候显得特别明显。为什么会这样呢？因为如果程序把字节直接写入视频显示存储器，它的执行速度要远快于不直接写入视频显示存储器。事实上，对于一些程序——比如要将图形显示在视频显示器上——操作系统就显得“心有余而力不足”，这就需要在操作系统之外另作处理。也许正是因为 MS-DOS 系统卓越的“反传统性”，使得大多数程序员都很热衷于它，它可以让程序员编写的程序最大限度地达到硬件的最快速度，更加充分地发挥硬件的性能。

恰恰是这个原因，运行在 IBM PC 上的流行软件通常依赖于 IBM PC 这个硬件平台，换句话说，就是这些软件是根据 IBM PC 的硬件特点编制的。为了和 IBM PC 竞争，其他机器制造商不得不沿袭这些特点。如果不这样做的话，再流行的软件也将流行不起来，因为它不能在这些机器上运行。所以在软件的显著位置往往会出现“与 IBM PC 百分之百兼容”的字样，这同时体现了软件对硬件的要求。

微软于 1983 年 3 月发布了 MS-DOS 2.0 版本，与最开始的版本相比，它加强了对硬盘驱动器的管理。虽然当时的硬盘容量很小（按今天的标准），但是发展很快，没经过多久，硬盘的容量变得越来越大。这带来了不少问题，硬盘容量越大存储的文件也就越多，存储的文件越多，当然，查找某个指定的文件或组织文件也就越困难。

为了解决上述问题，MS-DOS 2.0 引入了层次文件系统（hierarchical file system），它只是在原有 MS-DOS 的文件系统上做了一些小的改动。前面介绍过，目录存储在磁盘中特定区域，它是一个文件列表，包含了文件存储在磁盘位置的信息。在层次文件系统中，有些文件其本身可能就是目录，也就是说这些文件包含其他文件，其中的一些可能还是目录。在磁盘中，目录的称法也是有讲究的，常规的目录称为根目录（root directory），子目录（subdirectories）是包含在其他目录里的目录。有了目录（有时称文件夹，folder），就可以很方便地对相关文件进行分组，所以目录在磁盘文件的管理中起着非常重要的作用。

讲到操作系统，我们不能不提到著名的 UNIX 系统，它在操作系统的发展史上有着举足轻重的地位。实际上，层次文件系统和 MS-DOS 2.0 的其他很多功能都是从 UNIX 操作系统借鉴而来的。UNIX 操作系统是 20 世纪 70 年代初在贝尔实验室开发的，肯·汤普森（Ken Thompson，生于 1943 年）和丹尼斯·里奇（Dennis Ritchie，生于 1941 年）是此系统的主要开发者。UNIX 系统（包括名字本身）的发展历程在这里要提一下，UNIX 系统来源于早期的 Multics（Multiplex Information and Computing Services，表示多路复用信息和计算业务），Multics 是贝尔实验室和 MIT（麻省理工大学）和 GE（通用电气公司）合作开发的一个项目，开发初期由于 Multics 没能达到预定的目的而且进度缓慢，加之昂贵的开发代价，1969 年贝尔实验室退出了该项目，但肯·汤普森和丹尼斯·里奇继续对此进行了开发，为了区别于 Multics，取名为 UNIX。

对于计算机核心程序开发的精英们来说，UNIX 无疑是最受欢迎的操作系统。以前大部分操作系统是针对特定的计算机硬件平台开发的，但 UNIX 打破了常规，它不针对具体的计算机硬件平台，具有很好的可移植性（portable），也就意味着它在各种机器上都可以运行。

在开发 UNIX 的时候，贝尔实验室是电信业巨头 AT&T（American Telephone & Telegraph，美国电话电报公司）旗下的一员，为了限制 AT&T 在电话业务的垄断地位，法院裁定禁止 AT&T 销售 UNIX 系统，AT&T 被迫将它授权给别人。因此从 1973 年初，很多大学、公司和政府机构被授权使用和研究 UNIX，这在一定程度上促进了 UNIX 的发展。1983 年，AT&T 获准重返计算机业务并发布了它自己的 UNIX 版本。

由于 UNIX 的广泛授权导致了许多不同版本共存的情况，它们使用着不同的名字，运行在不同的计算机上，并由不同的经销商销售。由于 UNIX 的影响力和开源性，使得很多人将精力投入到 UNIX 中并推动着它的不断发展。当人们向 UNIX 中添加新的组件时，无形之中流行着一种不成文的“UNIX 思想”。在这种思想的指导下，人们都使用文本文件作为公用文件形式。许多 UNIX 实用程序读取文本文件，利用它提供的一些功能做些处理，然后将其写入另一个文本文件。因此可以用链的形式来组织 UNIX 实用程序，然后对这些文本文件做相应的处理。

在 20 世纪 60-70 年代，计算机不仅体积庞大而且价格昂贵，仅仅为一个人服务显然不太现实，为了能够让多个人同时使用计算机，必须有相应操作系统来支持，这也就是

开发 UNIX 系统的最初目的。使用 UNIX 系统的计算机通过时分复用（time sharing）技术——这种技术允许多个用户同时与计算机进行交互——来达到这个目的。计算机连接多个配备了显示器和键盘的终端（terminals），每个用户通过这些终端访问计算机。通过在所有终端间的快速切换，使用户感觉这台计算机似乎只为自己工作，而其实计算机同时在为多个用户服务。

如果在一个操作系统上可以同时运行多个程序，则称此系统为多任务（multitasking）操作系统。显然，与 CP/M 和 MS-DOS 这样的单任务的系统相比，这种操作系统要复杂得多。正是由于支持多任务这种功能，文件系统变得很复杂，因为同一个文件可能被多个用户同时访问。程序的运行需要占用内存空间，多任务系统就要考虑内存的分配问题，也就是说需要进行内存管理（memory management）。也许你会有疑问，多道程序并行运行需要占用大量内存，如果内存不够怎么办？为此，操作系统引入了虚拟内存（virtual memory）技术。虚拟内存是指，在磁盘上划出部分空间用做保存临时文件，程序把暂时不需要用的内存块放到临时文件里，待需要时再把它调入内存。。

UNIX 能够存在并发展到现在是无数人共同努力的结晶，如今 FSF（Free Software Foundation，自由软件基金会）和 GNU 项目为推动 UNIX 的发展注入了新的活力，它们都是由理查德·斯塔门（Richard Stallman）创建的。GNU 意味着："GNU 与 UNIX，既要划清界限又相辅相成"。GNU 项目的宗旨是：创建一个与 UNIX 系统兼容，但不受私有权限制的操作系统和开发环境。在这个项目的推动下，涌现出了许多和 UNIX 兼容的实用程序和工具，其中最著名的要算 Linux 系统。Linux 系统的内核（Core 与 Kernel 都可以表示内核的意思）和 UNIX 的是完全兼容的，它的大部分程序是由芬兰的李纳斯·托沃兹（Linus Torvalds）完成的。由于 Linux 系统的开源性，近年来已成为非常流行的操作系统。

从 20 世纪 80 年代中期开始，开发大型的、复杂度更高的系统成为了操作系统发展的一大趋势，例如苹果公司的 Macintosh 系统和微软的 Windows 系统，它们融合了图形和高级可视化视频显示技术，使应用程序变得更易于使用。关于图形和可视化方面的发展趋势，我们将在本书的最后一章进一步介绍。

23 定点数和浮点数

数字就是数字，整数、分数以及百分数等各种类型的数字与我们形影不离，它几乎出现在我们生活的所有角落。例如你加班 2.75 小时，而公司按正常工作时间的 1.5 倍支付你工资，你用这些钱买了半盒鸡蛋并交了 8.25%的销售税。就算你不是数字研究方面的专家，也可能对这种“数字生活”非常熟悉。我们还经常听到类似这样的统计信息“美国每个家庭的平均人口是 2.6 人”，但谁也不会为了满足这个数字表达的真实含义而把人拆解掉（这种事情想起来都觉得恐怖）。

在计算机存储器中，整数和分数之间的转换并不是这么随意。现在我们应该清楚，计算机中的一切数据都是以位的形式存储的，这就意味着所有的数都表示为二进制形式。但另一个不可否认的事实是，某些类型的数比其他类型更容易用位的形式来表示。

我们将从整数的二进制表示开始，这里的整数被数学家称做“自然数”（positive whole numbers），即计算机程序员口中的“正整数”（positive integers），之后将介绍如何利用 2 的补数来表示“负整数”（negative integers），该方法可以让正数和负数的相加变得非常简单。下面的表格列出了 8 位、16 位、32 位二进制数所能表示的正整数及其 2 的补数的范围。

数的位数	正整数的范围	整数的 2 的补数范围
8	0 ~ 255	−128 ~ 127
16	0 ~ 65,535	−32,768 ~ 32,767
32	0 ~ 4,294,967,295	−2,147,483,648 ~ 2,147,483,647

我们所要介绍的整数部分就是这些。除此之外，数学家还定义了用两个整数的比值表示的一类数，称做有理数（rational number）或分数（fraction）。例如，3/4 是一个有理数，因为它是整数 3 和 4 的比。我们也可以把 3/4 表示成十进制小数的形式，即 0.75。尽管我们可以把它写成十进制数的形式，但它实际上代表一个分数，即 75/100。

如第 7 章所述，在十进制数字系统中，小数点左边的数的每一位都和 10 的正整数次幂相关，而其右边的数的每一位都和 10 的负整数次幂相关。我们在第 7 章用到了 42705.684 这个实例，现在，我们把它表示为以下等价形式：

$$
\begin{aligned}
&4 \times 10000 + \\
&2 \times 1000 + \\
&7 \times 100 + \\
&0 \times 10 + \\
&5 \times 1 + \\
&6 \div 10 + \\
&8 \div 100 + \\
&4 \div 1000
\end{aligned}
$$

上面的除号表达了该位置的数与 10 的负整数次幂相关的情况，用下面的表达方式可以不用除号：

$$
\begin{aligned}
&4 \times 10000 + \\
&2 \times 1000 + \\
&7 \times 100 + \\
&0 \times 10 + \\
&5 \times 1 + \\
&6 \times 0.1 + \\
&8 \times 0.01 +
\end{aligned}
$$

$$4 \times 0.001$$

最后，我们将该数表示为 10 的幂的形式：

$$
\begin{aligned}
&4\times10^4 + \\
&2\times10^3 + \\
&7\times10^2 + \\
&0\times10^1 + \\
&5\times10^0 + \\
&6\times10^{-1} + \\
&8\times10^{-2} + \\
&4\times10^{-3}
\end{aligned}
$$

有一些有理数很难表示成小数，最明显的一个例子是 1/3，如果用 3 来除 1，会得到这样的结果：

$$0.33333333333333333333333333333333333333333333333333333333\ldots$$

在小数点的后面将会有无数个 3。通过在第一个 3 的上面加一条短横线，可以将这个无限循环的数简单表示为：

$$0.\overline{3}$$

尽管如此，把 1/3 表示为小数还不是很方便，但它毕竟是一个有理数，因为在本质上它是两个整数的比。类似的，1/7 可以表示为：

$$0.142857142857142857142857142857142857142857142857142857\ldots$$

或

$$0.\overline{142857}$$

无理数（irrational number）是一些更加奇特的数，如 2 的平方根等。它们不能表示为两个整数的比，这就意味着其小数部分是无穷的，而且毫无规律，没有循环，例如：

$$\sqrt{2} \approx 1.41421356237309504880168872420969807856967187537695\ldots$$

下面这个数学方程式的正数解就是 2 的平方根：

$$x^2-2=0$$

如果某个数不是任何以整数为系数的代数方程的解，那么这个数称做超越数（transcendental，所有的超越数都是无理数，但是反之不成立）。π就是一个典型的超越数，它是圆的周长与其直径的比值，可以近似的表示为：

3.1415926535897932846264338327950288419716939937511...

e是另一个典型的超越数，它是数学表达式：

$$(1+\frac{1}{n})^n$$

当 n 趋向无穷大时的值，近似的值为：

2.71828182845904523536028747135266249775724709369996...

目前我们所讨论过的所有数——有理数和无理数——统称为实数（real numbers）。使用实数定义它们的目的是为了将其与虚数（imaginary numbers）区别开来，虚数是负数的平方根。实数和虚数一起构成了复数（complex numbers）。不管名称如何，它们都有重要的作用，例如，虚数确实存在于现实世界，它在解决电子学的某些高级问题中有重要应用。

我们习惯于把数字看做连续（continuous）的，任意给出两个有理数，都可以找出一个位于它们之间的数。实际上，只需要取这两个数的平均值即可。但是，数字计算机对连续数据却无能为力，因为二进制中的每一位非0即1，两者之间没有任何数。这一特点决定了数字计算机只能处理离散（discrete）数据。二进制数的位数直接决定了所能表示的离散数值的个数。例如，如果你选择的二进制位数是32，则所能表示的自然数的范围是0～4,294,967,295。如果想要在计算机中存储4.5这个数，则需要选择新的方法并做一些其他方面的改进。

小数也可以表示为二进制数吗？当然可以，最简单的方法可能就是使用BCD码（二进制编码的十进制数）。如第19章所述，BCD码是将十进制数以二进制的形式进行编码，0～9之间的每一个数都需要用4位来表示，如下表所示。

十进制数字	BCD 代码
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

BCD 编码在程序处理用美元和美分表示的钱款、账户时特别有用。银行和保险公司是非常典型的两类整日与钱打交道的机构，这些机构所使用的计算机程序中，大多数小数所占用的存储空间仅仅相当于两个十进制数所占用的位数。

通常把两个 BCD 数字存放在一个字节，这种方式称为压缩 BCD（packed BCD）。由于 2 的补数不和 BCD 数一起使用，因此压缩 BCD 通常需要增加 1 位用来标识数的正负，该位被称做符号位（sign bit）。用一整个字节保存某个特定的 BCD 数是很方便的，但要为这个短小的符号位牺牲 4 位或 8 位的存储空间。

让我们来看一个例子，假设计算机程序所要处理的钱款数目在+/−100 万美元之间，这就意味着，需要表示的钱的数目的范围是 −9,999,999.99～99,999,999.99，因此保存在存储器中的每一笔钱的金额都需要 5 个字节。因此，−4,325,120.25 可以表示为下面 5 个字节：

00010100　00110010　01010001　00100000　00100101

将每个字节转换成十六进制数，上面的数可以等价地表示成：

14h　　32h　　51h　　20h　　25h

注意，最左边的半个字节所构成的 1 用来指明该数是负数，这个 1 即符号位。如果这半个字节所构成的数是 0，则说明该数是正数。组成该数的每一个数字都需要用 4 位来表示，从十六进制的表示形式中可以很直观地看到这一点。

如果要表示的数的范围扩大到 −99,999,999.99～99,999,999.99，则我们需要用 6 个字

节来实现，其中 5 个字节用来表示 10 个数字，另一个字节整个用来做符号位。

这种基于二进制的存储和标记方式也被称作定点格式(fixed-point format),所谓的“定点”是指小数点的位置总是在数的某个特定位置——在本例中，它位于两位小数之前。值得注意的是，有关小数点位置的计数信息并没有与整个数字一起存储。所以，使用定点小数的程序必须知道小数点的位置。你可以设计有任意小数位的定点小数，并且可以在程序中混合使用它们，但程序中对这些数进行算术运算的部分都需要知道小数点的位置，这样才能正确地对其做各种运算处理。

如果可以确定程序用到的数字不会大到超过预定的存储空间，并且这些数的小数位不会很多，那么使用定点格式的小数将是一个很好的选择。在表示非常大或非常小的数时，使用定点格式数是绝对不合适的。假设需要保留一块内存空间用来存放以英尺为单位的距离数据，可能存在的问题是某些距离的长度可能超出范围。地球与太阳之间的距离是 490,000,000,000 英尺，而氢原子的半径只有 0.000,000,000,26 英尺，如果采用定点格式的存储方案，为了存储这些极大或极小的数需要 12 个字节。

科学家和工程师们喜欢使用一种称为“科学计数法”(scientific notation) 的方法来记录这类较大或较小的数，利用这种计数系统可以更好地在计算机中存储这些数。科学计数法把每个数表示成有效位与 10 的幂的乘积的形式，这样就可以避免写一长串的 0，因此这种计数方式特别适合表示极大或极小的数。采用科学计数法，下面这个数：

$$490,000,000,000$$

可以记为：

$$4.9\times10^{11}$$

而

$$0.00000000026$$

可以表示为：

$$2.6\times10^{-10}$$

在上面的两个例子中，4.9 和 2.6 被称做小数部分或者首数 (characteristic)，有时候

也被称作尾数（mantissa，这个词通常与对数运算一起使用）。在计算机术语中这一部分被称做有效数（significand），因此为了保持一致，这里把科学计数法表示形式中的这一部分也称作有效数。

采用科学计数法表示的数可以分为两部分，其中指数（exponent）部分用来表示 10 的几次幂。在第一个例子中，指数是 11，第二个例子中指数是−10。指数可以表明小数点相对于有效数移动的距离。

为了便于操作，一般规定有效数的取值范围是大于或等于 1 而小于 10。尽管下面列出的各种写法表示的都是同一个数：

$$4.9\times10^{11}=49\times10^{10}=490\times10^{9}=0.49\times10^{12}=0.049\times10^{13}$$

但是上面等式中的第一种写法是最恰当的。这种写法有时被称做科学计数法的规范化式（normalized）。

这里需要说明，指数的正负性只是表明了数的大小，它并不能指明数本身的正负性。下面列出两个负数的科学计数法表示形式：

$$-5.8125\times10^{7}$$

与−58,125,000 相等。而

$$-5.8125\times10^{-7}$$

则与

$$-0.000,000,058,125$$

相等。

在计算机中，对于小数的存储方式，除了定点格式外还有另外一种选择，它被称做浮点格式（floating-point notation）。因为浮点格式是基于科学计数法的，所以它是存储极大或极小数的理想方式。但计算机中的浮点格式是借助二进制数实现的科学计数法形式，因此我们首先要了解如何用二进制表示小数。

实际操作起来比预想的要简单。在十进制数中，小数点右边的数字与 10 的负整数次

幂相关联；而在二进制数中，二进制小数点（就是一个简单的句点，看起来同十进制小数点一样）右边的数字和 2 的负整数次幂相关。例如，下面这个二进制数：

101.1101

可以用如下方式转换为十进制数：

$$
\begin{array}{l}
1\times4+\\
0\times2+\\
1\times1+\\
1\div2+\\
1\div4+\\
0\div8+\\
1\div16
\end{array}
$$

将乘数和除数用 2 的整数次幂替换，就可以替换除号：

$$
\begin{array}{l}
1\times2^{2}+\\
0\times2^{1}+\\
1\times2^{0}+\\
1\times2^{-1}+\\
1\times2^{-2}+\\
0\times2^{-3}+\\
1\times2^{-4}
\end{array}
$$

2 的负整数次幂等于从 1 开始反复除以 2，上式可以改写为：

$$
\begin{array}{l}
1\times4+\\
0\times2+\\
1\times1+\\
1\times0.5+\\
1\times0.25+\\
0\times0.125+\\
1\times0.0625
\end{array}
$$

经过这种计算，101.1101 与十进制数 5.8125 是相等的。

在十进制的科学计数法中，规范化式的有效数应该大于或等于 1 且小于 10；类似的，

在二进制的科学计数法中，规范化式的有效数应该大于或等于 1 且小于 10（即十进制的 2）。因此，在二进制的科学计数法中，下面这个数字：

$$101.1101$$

其规范化式应该是：

$$1.011101 \times 2^2$$

这个规则暗示了这样一个有趣的现象：在规范化二进制浮点数中，小数点的左边通常只有一个 1，除此之外没有其他数字。

当代大部分计算机和计算机程序在处理浮点数时所遵循的标准是由 IEEE（Institute of Electrical and Electronics Engineers，美国电气和电子工程师协会）于 1985 年制定的，ANSI（American National Standards Institute，美国国家标准局）也认可该标准。ANSI/IEEE Std 754-1985 称作 IEEE 二进制浮点数算术运算标准（IEEE Standard for Binary Floating-Point Arithmetic）——它只有 18 页——相对于其他标准来说是非常简短了，但却奠定了以简便方式编码二进制浮点数的基石。

IEEE 浮点数标准定义了两种基本的格式：以 4 个字节表示的单精度格式和以 8 个字节表示的双精度格式。

让我们首先来了解一下单精度格式。它的 4 个字节可以分为三个部分：1 位的符号位（0 代表正数，1 代表负数），8 位用做指数，最后的 23 位用做有效数。下表给出了单精度格式的三部分的划分方式，其中有效数的最低位在最右边。

$s = 1$ 位符号	$e = 8$ 位指数	$f = 23$ 位有效数

三部分共 32 位，也就是 4 个字节。我们刚才提到过，对于二进制科学计数法的规范化式，其有效数的小数点左边有且仅有一个 1，因此在 IEEE 浮点数标准中，这一位没有分配存储空间。在该标准中，仅存储有效数的 23 位小数部分，尽管存储的只有 23 位，但仍然称其精度为 24 位。我们将在下面的内容里体会 24 位精度的含义。

8 位指数部分的取值范围是 0～255，称为偏移（biased）指数，它的意思是：对于有符号指数，为了确定其实际所代表的值必须从指数中减去一个值——称做偏移量（bias）。对于单精度浮点数，其偏移量为 127。

指数 0 和 255 用于特殊的目的，稍后将简单介绍。如果指数的取值范围是 1～254，那么对于一个特定的数，可以用 s（符号位），e（指数）以及 f（有效数）来描述它：

$$(-1)^s \times 1.f \times 2^{e-127}$$

–1 的 s 次幂是数学上所采用的一种巧妙的方法，它的含义是：如果 $s=0$，则该数是正的（因为任何数的 0 次幂都是 1）；如果 $s=1$，则该数是负的（因为–1 的 1 次幂等于–1）。

表达式的中间部分是 $1.f$，其含义是：1 的后面是小数点，小数点后面跟着 23 位的有效数。$1.f$ 与 2 的幂相乘，其中指数等于内存中的 8 位的偏移指数减去 127。

注意，目前为止我们还没有学习如何表达那个经常遇到却又总被遗忘的一个数字：那就是“0”。这是一种特殊的情况，下面我们对其进行说明。

- 如果 $e=0$ 且 $f=0$，则该数为 0。在这种情况下，通常把 32 位都设置为 0 以表示该数为 0。但是符号位可以设置为 1，这种数可以解释为负 0。负 0 可以用来表示非常小的数，这些数极小以至于不能在单精度格式下用数字和指数来表示，但它们仍然小于 0。
- 如果 $e=0$ 且 $f\neq 0$，则该数是合法的，但不是规范化的。这类数可以表示为：

$$(-1)^s \times 0.f \times 2^{-127}$$

  注意，在有效数中，小数点的左边是 0。
- 如果 $e=255$ 且 $f=0$，则该数被解释为无穷大或无穷小，这取决于符号位 s 的值。
- 如果 $e=255$ 且 $f\neq 0$，则该值被解释为“不是一个数”，通常被缩写为 NaN（not a number）。NaN 用来表示未知的数或非法操作的结果。

单精度浮点格式下，可以表示的规格化的最小正、负二进制数是：

$$1.00000000000000000000000_2 \times 2^{-126}$$

小数点后面跟着 23 个二进制 0。单精度浮点格式下，可以表示的规格化的最大正、负二进制数是：

$$1.11111111111111111111111_2 \times 2^{127}$$

在十进制下，这两个数近似地等于 $1.175494351\times10^{-38}$ 和 3.402823466×10^{38}，这也就是

单精度浮点数的有效表示范围。

如前所述，10 位二进制数可以近似地用 3 位十进制数来表示。其含义是，如果把 10 位都置为 1，即十六进制的 3FFh 或十进制的 1023，它近似等于把十进制数的 3 位都置为 9，即 999，可以表示为下面的约等式：

$$2^{10} \approx 10^3$$

两者之间的这种关系意味着：单精度浮点数格式存放的 24 位二进制数大体上与 7 位的十进制数相等。因此，可以说单精度浮点格式提供 24 位的二进制精度或者 7 位的十进制精度。其深层的含义是什么呢？

当我们查看定点数时，其精确度是很明显的。例如，当我们表示钱款时，采用两位定点小数就可以精确到美分。但是对于采用浮点格式的数，就不能如此肯定了。其精确度依赖于指数的值，有时候浮点数可以精确到比美分还小的单位，但有时候其精确度甚至达不到美元。

这样说可能更合适：单精度浮点数的精度为$1/2^{24}$，或 1/16777216，或百万分之六，但其真正的含义是什么呢？

首先，这意味着在单精度浮点格式下，16,777,216 和 16,777,217 将表示成同一个数。不仅如此，处于这两个数之间的所有的数（例如，16,777,216.5）也将被表示成同一个数。所以上面提到的 3 个十进制数都按 32 位单精度浮点数：

4B800000h

来存放。将该数按符号位、指数位和有效数位划分，可以表示为：

0 10010111 00000000000000000000000

也就是：

$$1.00000000000000000000000_2 \times 2^{24}$$

下一个二进制浮点数可表示的最大有效数是 16,777,218，即：

$$1.00000000000000000000001_2 \times 2^{24}$$

以同一个浮点数来表示两个不同的十进制数，有时可能成为一个问题，也可能不会。

但如果你为银行编写程序，用单精度浮点数来存放以美元、美分为单位的数字时，就会发现 262144.00 美元和 262144.01 美元在计算机中存储为同一个数：

$$1.00000000000000000000000_2 \times 2^{18}$$

这也是为什么人们在处理以美元、美分表示的钱款数目时更愿意使用定点数的一个原因。当使用浮点数时，你会发现它还存在着一些让人崩溃的小问题。你的程序进行了一系列计算，应该得到的结果为 3.50 的，但由于使用浮点数，你得到的可能是 3.499999999999。这种问题在浮点数运算中经常发生，而且没有一套完整的解决方案。

如果想在程序中使用浮点格式数，但使用单精度格式又会出现各种问题，这时你可以考虑使用双精度浮点数（double-precision floating-point format）。这种类型的数需要用 8 个字节来表示，它的结构如下表所示。

$s = 1$ 位符号位	$e = 11$ 位指数位	$f = 52$ 位有效数

双精度浮点数的指数偏移量是 1023，或十六进制的 3FFh，因此以该格式存储的数可以表示为：

$$(-1)^s \times 1.f \times 2^{e-1023}$$

上面提到的关于单精度浮点格式下的 0，无穷大（小）和 NaN 的判断规则同样适用于双精度浮点格式。

双精度浮点格式下可以表示的最小正数或负数为：

$$1.0000000000000000000000000000000000000000000000000000_2 \times 2^{-1022}$$

注意，小数点的后面共有 52 个 0。同样的，可以表示的最大数为：

$$1.1111111111111111111111111111111111111111111111111111_2 \times 2^{1023}$$

其所能表示的范围，用十进制可以近似记为：

$$2.2250738585072014 \times 10^{-308} \sim 1.7976931348623158 \times 10^{308}$$

10 的 308 次幂是一个非常巨大的数，在 1 的后面跟着 308 个 0。

双精度浮点格式的有效数有 53 位（包括前面没有列出的那一位），大致相当于十进制的 16 位。与单精度浮点格式相比，这已经有了很大的改进了，但仍然不能避免两个不同的数存储为同一个结果的情况。例如，140,737,488,355,328.00 和 140,737,488,355,328.01 在内存中存放时，会被当做同一个数来处理，它们的双精度浮点格式表示为：

42E0000000000000h

即：

$$1.0000000000000000000000000000000000000000000000000000_2 \times 2^{47}$$

当然，为浮点数发明一种在内存中的存储方式，这只是在汇编程序使用浮点数所涉及的工作的一小部分。如果你决定闭门造车，完全独立地开发一台计算机，则必须要独立编写用于浮点数加、减、乘、除运算的函数集。幸运的是，有了前面关于整数四则运算的学习，这些关于浮点数的运算就可以分解成许多小的关于整数的加、减、乘、除运算，这样就能将问题大大简化。

例如，浮点数加法中最重要一点的就是如何对有效数相加，为了能使它们的有效位匹配，需要利用指数来确定对其如何移位。假设要进行下面的加法运算：

$$(1.1101 \times 2^5) + (1.0010 \times 2^2)$$

你需要把有效数部分的 11101 和 10010 相加，但并不是简单地直接相加。两个数的指数部分的不同决定了第二个数必须相对于第一个数右移。实际上我们要做的整数加法应该是 11101000 加 10010。最后的运算结果是：

$$1.1111010 \times 2^5$$

前面我们曾列出过太阳与地球的距离以及氢原子的半径，如果把这两个数相加会是怎样的结果呢？显而易见，因为它们两者的指数相差太大，因此较小的数对结果甚至没有影响。

两个浮点数的乘法意味着要把有效数当作整数相乘，并且把指数部分相加。为了使结果规范化，一般需要对指数调整一到两次。

浮点数运算另一层次的复杂性体现在处理一些较为繁杂的函数运算，例如平方根、

指数、对数和三角函数。但所有的这些运算都可以通过加、减、乘、除这四种基本的浮点数运算来实现。

例如，三角函数中的 sin 函数可以通过下面的一系列展开式近似计算：

$$\sin(x)=x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\cdots$$

参数 x 的值必须是弧度，360° 对应的弧度范围是 2π。上式中的感叹号表示阶乘运算符，它的意义是把从 1 到该数之间的所有整数相乘，例如 $5!=1\times2\times3\times4\times5$ 。这只是简单的乘法运算，每一项的指数部分也是乘法运算。其余的部分也只是简单的除法，加法或减法运算，这些都是容易实现的。上面的算式中，唯一让人感到棘手的地方是最后的省略部分，这意味着计算会一直继续下去。然而事情并没有想象中的那么糟糕，在实际运算中，如果把弧度的取值限制在 0～π/2 的范围内（从这个范围就可以推导出所有的正弦值），你根本不需要进行多少运算，因为大约展开 12 项后，就可以使结果精确到双精度浮点数要求的 53 位。

当然，使用计算机的目的就是帮助人们更加方便地解决问题，而编写程序来进行浮点数运算这一繁杂工作似乎和这个目的背道而驰。但这正是软件的优势所在：一旦某个人为特定的计算机编写了浮点数运算的程序，那么其他的人都可以使用它。浮点数运算在科学和工程类程序中极为重要，因此常常被赋予很高的优先级。在计算机发展的早期，为新制造的计算机做的第一项工作就是为其编写浮点数运算程序。

实际上，甚至可以直接利用计算机机器码指令来实现浮点数的运算。当然，实际做起来要比“动动嘴皮子”困难得多，但这也从另一个方面说明了浮点数运算的重要性。如果可以在硬件上实现浮点数算术运算——类似在 16 位微处理器上进行乘法和除法运算——则该机器上的所有的浮点数运算都会变得更快。

IBM 公司在 1954 年发布了 IBM 704，它是第一台将浮点数运算硬件作为可选配件的商用计算机，该机器以 36 位空间来存储所有的数。对于浮点数而言，其 36 位被分成 27 位的有效数，8 位的指数和 1 位的符号位。浮点运算硬件可以直接进行加法、减法、乘法和除法运算。其他的浮点运算则必须通过软件来实现。

从 1980 年开始，浮点运算硬件开始应用于桌面计算机，这起始于英特尔当年发布的

8087 数字协同处理（Numeric Data Coprocessor）芯片，当时这种集成电路被称做数学协同处理器（math coprocessor）或浮点运算单元（floating-point unit，FPU）。8087 不能独立工作，它只能与 8086 或 8088（Intel 的首个 16 位微处理器）芯片一起工作，因此被称做协处理器。

8087 拥有 40 个管脚，它使用的很多信号与 8086 或 8088 完全相同。微处理器和数学协处理器通过这些信号相连。当 CPU 读取到一条特殊指令 ESC（Escape）时——协同处理器开始接管控制权并执行下一条机器指令，该指令可以是三角函数、指数和对数等 68 条指令中的任一条。它所处理的数据类型遵循 IEEE 标准。在当时，8087 被认为是市面上最高水平的集成电路。

可以把协处理器当做一个小型的自包含计算机。当响应某个特定的浮点运算机器码指令时（例如 FSQRT 指令，它用来计算平方根），协处理器会以固有的方式执行存放在 ROM 中属于自己的指令序列。这些内部指令称做微代码（microcode）。通常，这些指令都是循环的，因此不能立即得到最终的结果。虽然如此，但数学协处理器在运算速度方面仍然表现优异，与软件方法相比，其速度至少是后者的 10 倍。

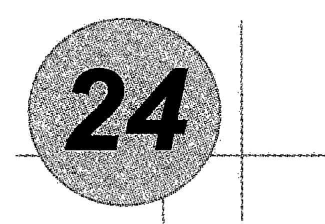
在最初版本的 IBM PC 主板上，位于 8080 芯片的右边有 1 个 40 个管脚的插槽供 8087 芯片接入。但令人失望的是，这个插槽是空的，用户如果需要进行浮点数运算就必须单独购置一块 8087 芯片，并将其插入主板后才能使用。安装数学协处理器并不能提高所有应用程序的运行速度，因为有些应用程序——比如文字处理程序——几乎用不到浮点数运算。其他应用程序，比如电子表格处理程序，对浮点数运算依赖程度很高，在安装了数学协处理器之后，它们的执行速度有很大的提高，但并非所有的程序都是如此。

可以看到，在安装了数学协处理器后，程序员必须使用协处理器机器码指令来编写特殊的代码，因为数学协处理器不是标准硬件，因此它只能执行这些特殊的代码。而这些工作让程序员烦不胜烦。尽管他们不愿意，他们仍不得不编写自己的浮点数运算子程序（因为大多数人并没有安装数学协处理器），因此这就多了一个额外的工作——一个并不轻松的工作——在程序中支持 8087 芯片。最后就出现了这样的局面：如果机器上安装了数学协处理器，程序员就要学会编写相应的应用程序以支持它的运行；如果没有安装，程序员就要通过编程来模拟它进行浮点数的运算。

在随后的几年内，英特尔还发布了与 286 配合工作的 287 数学协处理器，与 386 配

合工作的 387 数学协处理器。但是，在 1989 年发布的 486DX 芯片中，FPU 已经内建在 CPU 的结构里，它不再作为一个配件供选择安装了。令人失望的是，在 1991 年发布的一款低端芯片 486SX 中，英特尔没有为该其内建 FPU，而是提供了一块可选的 487SX 数学协处理芯片。但 1993 年发布的奔腾芯片中，CPU 内置 FPU 再次成为标准，也许这是永远的标准。在 1990 年发布的 68040 芯片中，摩托罗拉首次将 FPU 集成在 CPU 中，在此之前，摩托罗拉发布了 68881 和 68882 数学协处理器来支持 68000 家族早期的微处理器。PowerPC 芯片同样使用了内置浮点数运算硬件的技术。

浮点数运算硬件对于困惑的汇编程序员来说无疑是个惊喜的礼物，但相对于 20 世纪 50 年代开始的某些其他工作来言，这只是历史所迈出的一小步。接下来，我们的探索之旅即将到达下一站：计算机语言。



高级语言与低级语言

使用机器码编写程序就如同用牙签吃东西，伸出手臂使出较大的力气刺向食物，但每次都只获取到小小的一块，这个过程是辛苦且漫长的。同样的，每个机器码字节所能完成的工作，是你能想象到的最微小且最简单的工作——从内存获取一个数，之后加载到处理器，再把它与另一个数相加，最终将运算结果保存到内存等——正因如此，很难想象如何使用这些机器码构成一个完整的程序。

目前为止，至少对于在第 22 章讨论的原始模型阶段来说，我们已经取得了一定的进步，在那个阶段，我们使用过控制面板上的开关将二进制数据输入到存储器。在第 22 章中，介绍了如何编写一段简单的程序，让我们可以利用键盘将十六进制机器码输入计算机，以及通过视频显示设备来检查这些代码。这种改进固然可取，但仍不是我们的终极目标。

前面的章节介绍过，可以使用某些较短的助记符来关联机器码字节，这些助记符包括 MOV，ADD，CALL，HLT 等，通过这些类似英语的符号我们可以较方便地引用机器

码。通常这些助记符的后面会跟着操作数，这可以进一步指明它所关联的机器码指令的功能。例如 8080 机器码字节 46h，它的功能是令处理器将存储在内存特定地址的字节转移至寄存器 B，而该地址由寄存器对 HL 中的 16 位数寻址。这个操作可以简单地写做：

```
MOV B,[HL]
```

显然，使用汇编语言编写程序要比使用机器语言简单得多，但微处理器并不能解释汇编语言。在前面的章节中我们已经学习了如何在纸上编写汇编程序，但只有当你确实准备在微处理器上运行汇编程序，才会手工对其汇编，这样就可以将汇编语言程序的语句转换成了机器语言代码，并把它们输入内存。

当然，我们希望最好由计算机能独自完成语言转换的工作。如果你的 8080 计算机正在运行 CP/M 操作系统，而且你已经拥有了所有必需的工具，那就再好不过了，因为下面我们将介绍其工作原理。

第一步，建立一个文本文件，并将汇编语言程序输入到该文本文件中。这项工作可以使用 CP/M 的应用程序 ED.COM 来完成。该程序是一个可以用来创建、修改文本文件的编辑器。假设你把该文本文件命名为 PROGRAM1.ASM，其中 ASM 是文件类型，用来指明该文本文件的内容是由汇编语言程序组成。这个文件的内容如下：

```
        ORG 0100h
        LXI DE, Text
        MVI C,9
        CALL 5
        RET
Text:   DB 'Hello!$'
        END
```

这个文件中有两条语句我们从未接触过。第一条语句是 ORG（origin），它不与任何 8080 指令对应，其功能是用来指明下面语句的地址从 0100h 地址处开始。如前所述，该地址是 CP/M 将程序装入内存的起始地址。

第二条语句是 LXI（Load Extended Immediate）指令，其功能是将一个 16 位数加载到寄存器对 DE。在本例中，该 16 位数是由标记 Text 提供的。该标记在程序底端的附近，位于 DB（Data Byte）语句之前。DB 语句我们也是第一次遇到，其后可以跟着一些字节，这些字节以逗号分隔或者用单引号括起来（如本例）。

MVI（Move Immediate）语句将数值 9 转移到寄存器 C。CALL 5 语句实现 CP/M 的函数调用功能。函数 5 的作用是：显示以寄存器对 DE 给出的地址为起始处的字符串，直到遇到$结束（可以看到，在程序的结尾处使用了美元符号“$”作为文本的结束标志，这种方式看起来很奇怪，但 CP/M 就是采取的这种方式）。最后的 RET 语句用来结束程序，并把控制权交还给 CP/M（实际上，这只是结束 CP/M 程序的方法之一）。END 语句用来指明汇编语言文件已经结束。

现在我们已经有了一个包含 7 行语句的文本文件，下一步要做的就是对其进行汇编，即将其转换成机器语言代码。以前这项工作是通过手工完成的，但现在我们的机器运行的是 CP/M 系统，可以利用 CP/M 中一个叫做 ASM.COM 的模块来完成这项工作。该模块是 CP/M 的汇编器（assembler）。可以在 CP/M 的命令行中使用下面的语句运行 ASM.COM 文件：

```
ASM PROGRAM1.ASM
```

ASM 对 PROGRAM1.ASM 文件进行汇编，产生一个名为 PROGRAM1.COM 的新文件，PROGRAM1.COM 包含了与我们编写的汇编程序相对应的机器码（实际上，该过程还包含另一个步骤，但在该操作中并不重要）。现在就可以使用 CP/M 的命令行来运行 PROGRAM1.COM 文件，程序运行的结果是显示字符串“Hello!”然后结束。

PROGRAM1.COM 文件包含以下 16 个字节：

```
11  09  01  0E  09  CD  05  00  C9  48  65  6C  6C  6F  21  24
```

开始的 3 个字节是 LXI 指令，其后的两个字节是 MVI 指令，接下来的三个字节是 CALL 指令，紧随其后的一个字节是 RET 指令，最后的 7 个字节是 ASCII 码，包括 5 个字母“Hello”，感叹号“!”以及美元符号“$”。

像 ASM.COM 这样的汇编器程序所做的工作是：读取一个汇编语言文件（source-code，通常称做源代码文件），将其转换得到一个包含机器码的文件——可执行文件（executable file）。从宏观的角度来看，汇编器是非常简单的，因为构成汇编语言的助记符和机器码之间是一一对应的。汇编器拥有一张包括所有可能助记符及其参数的表，它逐行读取汇编语言程序，把每一行都分解成为助记符和参数，然后把这些短小的单词和字符与表中的内容匹配。通过这种匹配的过程，每一个语句都会找到与其对应的机器码指令。

注意，汇编器如何知道 LXI 指令必须将寄存器 DE 的值设置为地址 0109h（Text 的地址）。如果 LXI 指令本身被存放在地址 0100h 处（CP/M 将程序加载至内存开始运行时的起始地址），而 0109h 则是 Text 字符串的起始地址。一般来说，程序员在使用汇编器时有很多方便之处，其中一点就是不需要关心汇编程序各部分在内存中的存放地址。

第一个编写汇编器的人需要手工对程序汇编。如果要为机器写一个新的汇编器（或者对其修改），则可以使用汇编语言编写该程序，然后使用原有的汇编器对其汇编。一旦新的汇编器通过了汇编，则它也就可以对自身进行汇编。

每当一种新的微处理器面世，就需要为其编写新的汇编器。然而，新的汇编器可以在已有的计算机上编写，并利用其汇编器进行汇编。这种方式称为交叉汇编（cross-assembler），即利用计算机 A 的汇编器对运行在计算机 B 上的程序汇编。

虽然汇编器的引入消除了汇编语言编程中重复性的劳动部分（即手动汇编部分），但汇编语言仍然存在两个主要问题。第一个问题（也许你已经意识到了），使用汇编语言编程非常乏味，因为这是在微处理器芯片级的编程，因此不得不考虑每一个微小的细节。

汇编语言存在的第二个问题是不可“移植”（portable）。如果你为 Intel 8080 写了一个汇编语言程序，则该程序不能在 Motorola 6800 上运行，你必须在 6800 上重写一个相同功能的汇编语言程序。编写类似程序的过程也许没有编写第一个程序那么困难，因为你应经解决了程序的组织和算法问题，但仍然还有很多工作要做。

上一章介绍了现代微处理器集成浮点运算机器码指令的原理。不可否认，这已经为我们带来了很大的便利，但仍不能令人特别满意。一种更好的方式是：完全放弃那些实现每个基本操作的机器码指令，这些指令与处理器相关，因而导致程序缺乏移植性。我们采用的替代策略是使用一些经典的数学表达式来描述复杂的数学运算。下面是一个表达式的例子：

$$A \times Sin(2 \times \mathrm{PI} + B) / C$$

上式中的 A，B，C 代表数字，而 PI = 3.14159。

这看起来不错，为什么不动手尝试一下呢？假设在某个文本文件中有这样一个表达式，那么我们可以尝试编写一个汇编语言程序来读取该文本文件，并将其中的数学表达

式转换为机器码。

如果只需要计算一次该表达式，那么可以手工计算或借助计算器来完成。如果需要对 A、B、C 取不同的值多次计算该表达式，那么你可能要考虑使用计算机来完成这些计算。因此，代数表达式不会孤立地出现，必须考虑其前后的语句，这些语句使表达式对不同的值进行运算。

现在你所创建的东西已经触及所谓的高级程序设计语言（high-level programming language）。我们一直在介绍的汇编语言称做低级语言（low-level programming language），因为它与计算机硬件的关系相当紧密。尽管除了汇编语言以外的其他程序设计语言都可以称为“高级语言”，但它们之间还是有高低之分的，一些语言通常被认为比别的语言更高级。如果你是一家公司的总裁，坐在计算机前输入这些命令（也可能做得更轻松：口头发布这个命令），“计算出本年度的收益和损耗，生成年度报表，最后打印出 2000 份送至每个股东”，你所使用的才真正是一种非常高级的语言！但在实际工作中，程序设计语言还达不到这种理想化的水平。

人类语言通常都是经历了千百年复杂的互相影响、偶然演变以及不断吐故纳新才形成的，就算一些人工语言如世界语（Esperanto），也处处显露出与现实语言的渊源。但高级程序设计语言是经过深思熟虑的设计的，更加概念化的语言。设计程序设计语言所面临的一大挑战就是：如何让语言更具吸引力。因为语言定义了人们向计算机传送指令的方式，只有更易用的方式才能让人们对语言产生兴趣。据 1993 年的一项估算，从 1950 年到 1993 年大约有 1000 多种高级程序设计语言被发明出来并被应用。

然而，仅仅定义（define）高级语言，包括定义语言的语法（syntax）来表达该语言可以描述的一切事物，还远远不够；我们还需要为其编写一个编译器（compiler），编译器可以将高级语言的程序语句转换成机器码指令。同汇编器类似，编译器也是逐字逐句地读取源文件并将其分解成为短语、符号和数字的，但实现过程要比汇编器更加复杂。从某些方面来看，汇编器相对简单，因为汇编语言的语句和机器码是一一对应的。而一般的高级语言却不具备这种对应关系，编译器通常必须把一条语句转换多个机器码指令。编译器的编写非常复杂，许多书都是用全部的篇幅来讲解如何设计和构造编译器。

当然，任何事物都是具有两面性的，高级语言也不例外，它有很多优势但也存在不少缺陷。高级语言最基本的优点在于它比汇编语言易于学习并且更容易编写程序，用高

级语言编写的程序通常更加清晰简明——与汇编语言不同，高级语言通常不依赖于特定的处理器，因此它们通常具有良好的可移植性。因为这种特点，使用高级语言的程序员不再需要关心最终运行程序的计算机的底层结构。当然，如果要在不同类型的处理器上运行程序，则需要用处理器对应的编译器将程序转换成对应的机器码。因此，最后生成的可执行文件仍然只适用于特定的处理器。

另一方面，有一种普遍现象：一个优秀的汇编程序员所编写的程序比编译器所产生的代码更加有效率。也就是说，从高级语言程序生成的可执行程序比相同功能的汇编语言程序更大，并且运行速度更慢（但从近年的发展来看，这种差别已变得不再明显，因为微处理器变得更加复杂，而且编译器在优化代码方面也更加成熟）。

此外，虽然高级语言提高了处理器的易用性，但并没有让其变得更强大。微处理器的任何一个功能都可以通过汇编语言实现，因此汇编语言可以高度利用处理器的功能。因为高级语言必须转化成机器码，所以它只会降低微处理器的能力。事实上，如果某种高级语言具有真正意义的可移植性，那么它将不能使用某些处理器的特有功能。

例如，许多微处理器都有移位指令。如前所述，这些指令能将累加器中的字节的每一位向左或向右移动。但事实上，几乎没有哪一种高级语言包含这种操作。如果在程序中需要进行移位操作，则必须通过乘 2 或除 2 来模拟该过程（这并不是什么坏事：事实上，许多现代编译器都是利用处理器的移位指令来实现乘以或除以 2 的幂的）。除此之外，许多高级语言也不包括按位逻辑运算。

在早期的家用计算机中，大部分应用程序都是用汇编语言写的，而现在除了一些特殊的应用场合之外，汇编语言已经很少使用了。而今处理器引入了一些新的硬件，可以实现流水线技术——同时有若干个指令码渐次执行——这使得汇编语言变得更加复杂且不易处理。与此同时，编译器却变得更加成熟，越来越多的程序开始使用高级语言来编写。现代计算机大容量的存储器也作为一个重要的角色，推动了这种趋势：程序员不再局限于编写运行在小内存和小磁盘上的程序。

早期的计算机设计者都曾尝试用数学符号来描述问题，但公认的第一个真正可以工作的编译器是 A-0，它是为 UNIVAC 开发的编译器，于 1952 年由雷明顿兰德公司（Remington-Rand）的格瑞斯·穆雷·霍珀（Grace Murray Hopper，1906-1992）开发完成。霍珀博士的早期计算机研究工作始于 1944 年，那时她效力于霍华德·艾肯（Howard

Aiken），主要研究 Mark Ⅰ。在她八十多岁的时候，仍然孜孜不倦地在计算机界工作，当时她在 DEC（Digital Equipment Corporation）公司从事公关事务。

FORTRAN 语言是目前仍在使用的最古老的高级语言（虽然这些年来人们对其进行了大量修改）。你可能注意到了，很多计算机语言都是以大写字母命名的，这是因为它们的名字大都是由几个单词的首字母组成。FORTRAN 这个名字来源于 FORmula 的前三个字母和 TRANslation 的前四个字母的组合，它由 IBM 在 20 世纪 50 年代中期开发，主要应用于 704 系列计算机。自其发布的几十年来，FORTRAN 一直被认为是科学和工程应用程序开发的首选语言。它广泛地支持浮点运算，甚至支持非常复杂的数的运算（即我们上一章讲到的由实数和虚数构成的复数）。

任何一种计算机程序设计语言都有其支持者和批评者，而且人们通常只对自己喜欢的语言有热情。本书尽量以一种客观的态度来讨论某种语言，这里选取了一种语言作为原型，通过它来解释那些几乎已经销声匿迹的程序设计概念。我们的选择是 ALGOL（即 ALGOrithmic 的缩写，有趣的是，ALGOL 也是仙女座第二亮的恒星的名字）。ALGOL 作为过去 40 年中许多曾经流行一时的通用高级语言的直接鼻祖，也非常适合用来研究高级程序设计语言的本质，该语言可看做是一粒种子，它的成长最终形成了高级语言这棵大树。直到今天，人们仍然在使用"类 ALGOL"程序设计语言的概念。

ALGOL 语言的原版由某国际委员会在 1957 至 1958 年间设计，它被称做 ALGOL 58。两年后，也就是在 1960 年，ALGOL 58 的改进版 ALGOL 60 面世，其最终版本是 ALGOL 68。本章所采用的版本在 *Revised Report on the Algorithmic Language ALGOL 60* 说明文档中有具体描述，该文档于 1962 年完成并在 1963 年首次发行。

让我们开始写第一个 ALGOL 程序。假设我们使用的操作系统平台是 CP/M 或 MS-DOS，并且安装了一个名为 ALGOL.COM 的编译器。该程序是一个文本文件，命名为 FIRST.ALG。注意，文件类型名是 ALG。

ALGOL 程序以 begin 开始，以 end 作为结尾，程序的主要内容被包括在这两个语句之间。下面的程序用来显示一行文本：

```
begin
     print ('This is my fist ALGOL program!');
ende
```

通过在命令行运行 ALGOL 编译器对 FIRST.ALG 文件进行编译，其格式如下：

```
ALGOL FIRST.ALG
```

ALGOL 编译器对这条命令很可能做出这种响应，在显示设备上给出以下提示信息：

```
Line 3: Unrecognized keyword 'ende'.
```

ALGOL 编译器对拼写的检查非常严格，它在这一点上比传统的语文教师更甚。因为输入程序时，误把“end”拼写做“ende”，所以编译器通过提示信息告诉我们程序中有语法错误（syntax error）。当编译器检查到“ende”时，它期待能遇到一个可识别的关键字（keyword），但由于上述错误，编译不能通过。

将程序中的错误改正之后，可以再次执行编译命令。由于系统平台和编译器版本的不同，有时编译器会直接生成一个可执行文件（CP/M 平台下此文件名为 FIRST.COM，MS-DOS 平台下名为 FIRST.EXE）；有时还需要再执行一个步骤才可以完成。不论是哪种情况，最后你都可以在命令行执行 FIRST 程序：

```
FIRST
```

FIRST 程序会对此响应，并显示以下内容：

```
This is my fist ALGOL program!
```

注意，这里还有一个拼写错误：first 被误做 fist！编译器没有检查出这个错误，因此它被称为运行时错误（run-time error）——程序被执行时才出现的错误。

很明显，我们的第一个 ALGOL 程序中，print 语句的功能是把一些信息显示到屏幕上，在本程序中是显示一行文本（从功能的角度来看，该程序与本章开始所给出的汇编程序是等价的）。ALGOL 语言的正式规范中并不包括 print 语句，但我们假设所使用的特定 ALGOL 编译器包括这个便利的工具，它有时候也被称做内部函数（built-in function）。除了 begin 和 end 之外的大部分 ALGOL 语句都要以分号结尾。你可能注意到了 print 语句使用了向右缩进的格式，这并不是必要的，其作用只是为了让程序的结构更加清晰。

假设现在要编写一个用于两个数相乘的程序。每一种程序设计语言都包括变量（variable）的概念。程序中的变量可以是一个字母、一个短的字母序列，也可以是一个单词，由程序员自己决定。变量名实际上对应内存的一个存储单元，但在程序中是通过名

字来访问该存储单元的，而不是直接使用存储单元的地址值。下面的程序定义了三个变量，分别命名为 *a*，*b*，*c*:

```
begin
    real a, b, c;
    a := 535.43;
    b := 289.771;
    c := a × b;
    print ('The product of', a, ' and ', b, ' is ', c);
end
```

real 语句称为声明（declaration）语句，用来指明程序中要定义的变量。在该程序中，变量 *a*，*b*，*c* 被定义为实数（real）类型或浮点数类型（同时，ALGOL 语言也支持使用 integer 关键字来定义整数型变量）。程序设计语言中的变量名通常以字母开头，变量名也可以包括数字，但前提是第一个字符必须是字母。变量名不能含有空格，也不能包含除字母和数字以外的其他大部分字符。通常编译器会规定变量名的最大长度，本章用到的变量一律以单个字母命名。

假如我们使用的特定 ALGOL 编译器支持 IEEE 浮点数标准，则本程序中所定义的三个变量每一个需要 4 个字节的存储空间（采用单精度格式）或 8 个字节的存储空间（采用双精度格式）。

声明语句之后的三个语句是赋值（assignment）语句。在 ALGOL 语言中，赋值语句很容易被识别，因为它的格式很固定，总是在冒号后面跟着一个等号（在大多数计算机语言中，赋值语句通常只包括等号）。赋值语句的冒号左边是一个变量，而等号右边是一个表达式，表达式的计算结果将被赋值给左边的变量。前两条赋值语句指明，变量 *a*，*b* 将分别被赋予一个特定的值；第三条赋值语句指明，将 *a* 和 *b* 的乘积赋值给变量 *c*。

时至今日，我们所熟悉的乘法符号“×”已经不允许出现在程序设计语言中了，因为它没有被包括在 ASCII 和 EBDCIC 字符集中。大多数程序设计语言使用星号（*）来替代它作为程序中的乘号标记。尽管 ALGOL 使用了普遍使用的斜杠（/）作为除法标记，但在该语言仍然可以使用除法标记（÷），该标记用于整数除法，用来指明被除数与除数的倍数关系。ALGOL 还使用了另一个非 ASCII 字符“↑”，该箭头符号用来做乘方运算。

最后的 print 语句用来显示所有变量的值。它包含文本和变量，并以逗号分隔。print

语句的主要工作并不是用来显示 ASCII 码值的，但本程序中却做了更多的工作：将浮点数也转换成了 ASCII 码并显示：

```
The product of 535.43 and 289.771 is 155152.08653
```

接着会执行 end 语句，程序终止并将控制权交还给操作系统。

如果要将另外两个数相乘，则需要做以下工作：修改程序，改变变量的值，重新编译并重新运行程序，这将是一件非常烦琐的工作。为了避免这些重复工作，我们可以借助于另一个内部函数 read。修改后的程序如下：

```
begin
   real a, b, c;
   print ('Enter the first number: ');
   read (a);
   print ('Enter the second number: ');
   read (b);
   c := a × b;
   print ('The product of ', a, ' and ', b, ' is ', c);
end
```

read 语句的功能是读取从键盘键入的 ASCII 码值，并将其转换成浮点数。

循环（loop）是高级语言的重要组成部分。循环使得程序可以对同一个变量的不同取值反复执行相同的操作。假设我们要写一段程序用来计算 3，5，7，9 各自的平方，可以这样编写程序：

```
begin
   real a, b;

   for a := 3, 5, 7, 9 do
          begin
              b := a × a × a;
              print ('The cube of ', a, ' is ', b);
           end
end
```

for 语句将变量 *a* 的值第一次设为 3，然后执行 do 关键字后面的语句。如果 do 后面要执行的语句不止一条（如本例），则必须将它们置于 begin 和 end 之间，这两个关键字定义了一个语句块（block）。第一次循环之后，for 语句会一次为 *a* 赋值 5，7，9 并执行

相同的语句块。

下面的程序中采用了 for 语句的另一种使用方式，这段程序用来计算 3 ~ 99 之间所有奇数的立方。

```
begin
    real a, b;
    for a := 3 step 2 until 99 do
            begin
                 b := a × a × a;
                 print ('The cube of ', a, ' is ', b);
            end
end
```

for 语句将变量 *a* 初始化为 3，并执行 for 后面的语句块。第一次循环结束后，变量 *a* 与 step 关键字后面的增量相加，这里是 2。新得到的 *a* 的值是 5，它将用于第二次执行语句块。变量 *a* 继续增加 2 并用于下一次循环，直到 *a* 的值超过 99，这时 for 循环结束。

一般而言，程序设计语言对语法都有着非常严格的要求。在 ALGOL 60 中，就关键字 for 而言，其语法格式是：for 的后面只能跟一个变量名。而英语中的这种限制宽松的多，单词 for 的后面可以跟所有类型的单词，例如“for example”，“for can”等。尽管编译器是非常复杂的程序，但其所能解释的语言显然要比人类的语言简单得多。

大部分程序设计语言的另一个重要特征体现在条件（conditional）语句的使用。条件语句的特点是，只有当某个条件成立时才会执行另一条对应的语句。在下面的例子中，我们使用 ALGOL 的内部函数 sqrt 来计算一些数的平方根。sqrt 函数的参数不能是负数，因此要在程序中通过条件测试避免这种情况。

```
begin
    real a, b;

    print ('Enter a number: ');
    read (a);
    if a < 0 then
            print ('Sorry, the number was negative.');
    else
            begin
                 b = sqrt(a);
                 print ('The square root of ', a, ' is ', b);
```

```
            end
    end
```

左尖括号（<）是小于号。如果程序的使用者输入的是一个小于 0 的数，if 语句中的判断语句为真，因此第一个 print 语句将会被执行。反之，如果该数大于或等于 0，则 else 关键字后面的语句块则会被执行。

本章目前所用到的变量都是一个变量对应一个值，我们也可以用一个变量对应多个值，数组（array）就是一个很好的选择。在 ALGOL 程序中可以这样声明一个数组：

```
real array a[1:100];
```

该语句定义一个数组变量 ***a***，它可以用来存放 100 个不同的浮点数，这些数被称做数组元素。可以使用数组名加标号的方式来引用数组元素，例如，第一个数组元素是 ***a***[1]，第二个是 ***a***[2]，最后一个是 ***a***[100]。方括号中的数字称做数组下标（index）。

下面的程序用来计算 1 ~ 100 所有数的平方根，将结果保存在一个数组中，然后再通过循环将这些结果显示出来。代码如下：

```
begin
    real array a[1:100];
    integer i;
    for i := 1 step 1 until 100 do
            a[i] := sqrt(i);

    for i := 1 step 1 until 100 do
            print ('The square root of ', i, ' is ', a[i]);
end
```

程序中还定义了一个整型变量 ***i***（由于它是 integer 的首字母，经常被程序员用做整型变量名）。第一个 for 循环的执行过程中，每个数组元素被赋值为其下标的平方根；第二个 for 循环执行过程中，数组中的每一个元素被显示出来。

变量的类型有很多，除了我们已经介绍过的实型和整型之外，变量还可以被声明为布尔型（Boolean，该名称是为了纪念第 10 章提到的乔治・布尔）。布尔变量的取值只可能有两种，即 true 和 false。在本章的最后将介绍一个用到布尔数组的例子（这个例子也将用到目前所介绍的大部分内容），来实现一个寻找素数的著名算法——爱拉托逊斯筛法（Sieve of Eratosthenes）。爱拉托逊斯（约公元前 276–196 年）传说是亚历山大图书馆的管

理员，他因准确计算出地球的周长而永载史册。

素数是只能被 1 及其本身整除的一类整数。第一个素数是 2（也是唯一的偶数素数），其他的素数还包括 3，5，7，11，13，17，等等。

爱拉托逊斯方法以 2 开始的整数表开始，因为 2 是素数，因此所有可以被 2 整除的数都被排除掉（即除了 2 之外的全部偶数）。接下来是 3，因为 3 是素数，因此所有能被 3 整除的数也被排除掉。因为 4 在第一个步骤中已经被排除掉，所以下一个要考虑的数是 5，即排除所有 5 的倍数。按这种方式不断循环，最后剩下的都是素数。

下面的 ALGOL 程序用来筛选 2 ~ 10,000 之间的所有素数，程序中定义了一个布尔数组，用来对所有的数进行标识。该程序如下：

```
begin
   Boolean array a[2:10000];
   integer i, j;

   for i := 2 step 1 until 10000 do
   a[i] := true;

   for i := 2 step 1 until 100 do
         if a[i] then
                for j := 2 step 1 until 10000 ÷ i do
                      a[i × j] := false;

   for i := 2 step 1 until 10000 do
         if a[i] then
               print (i);
end
```

第一个 for 循环将数组 ***a*** 的每一个元素的初始值设置为布尔值 true。这里的 true 表示该位置的数是素数，因此现在程序默认所有的数都是素数。第二个 for 循环的范围是 1 ~ 100（100 刚好是 10000 的平方根）。在第二个 for 循环中，如果判断条件成立，该数为素数，即 ***a***[*i*]为 true，则第三个 for 循环则会把该数的所有小于或等于 10000 的倍数（除了其本身）设置为 false，因为这些数都不是素数。最后的 for 循环用来输出所有的素数，这里的判断条件是：若 ***a***[*i*]为 true，则 *i* 为素数。

程序设计到底是一门科学还是一门艺术呢？这的确是一个有趣的问题，一些人甚至

还为此争论不休：一方面，你或许在大学里系统地学习了计算机科学（Computer Science）课程；另一方面，你又读过如唐纳德·克努斯（Donald Knuth）的名著《计算机编程艺术系列》（*The Art of Computer Programming series*）等著作。然而物理学家理查德·费叶曼（Richard Feynman）曾这样写道："从某种程度上看计算机科学像是一种工程，它的工作范畴是利用一些事物去实现其他事物。"

在程序设计中有一种现象：如果让 100 个人来编写输出素数的程序，你可能会得到 100 个不同的解决方法。就算所有的程序员都使用"爱拉托逊斯筛法"来解决这个问题，其最后所写的程序也不一定与本文所写程序完全相同。如果说程序设计是一门科学，那么就不应该出现如此多的解法，而不正确的方法将会非常明显。偶尔，一个程序设计问题会诱发出极富创造性的火花或洞若观火般的觉察力，这就是所谓的程序设计的"艺术"。但是，程序设计的更多的时候是设计和建造，就像修建一座大桥的过程。

早期的程序设计对编程人员的要求很高，所以很多早期的程序员都是科学家或工程师，他们通常利用 FORTRAN 或 ALGOL 中的数学算法来描述并解决各自领域的问题。回顾程序设计语言发展的整个历程时，我们会发现，人们一直在努力开发一种能为更大范围的人群所使用的语言。

第一个成功地为商务系统所使用的程序设计语言是 COBOL（COmmon Business Oriented Language），今天它仍然被广泛使用。COBOL 于 1959 年开始开发，由美国工业界和国防部组成的委员会发起并实施，它的设计思路受到格瑞斯·霍珀早期编译器的影响。从某些方面来看，COBOL 的设计中渗透了这种思想：使管理人员——可能并不进行实际的编码工作——但他们至少可以看懂程序代码，而且能够检测程序能否完成预定工作（实际上这种情况非常少见）。

COBOL 语言广泛支持读取记录（record）和生成报表（report）。记录是按照统一方式归类整理的信息的集合。例如，保险公司一般会维护一个包括其所售的所有保险信息的大型文件，每一项保险业务称为一条单独的记录。每一条记录包括客户的姓名、出生日期等信息。早期编写的 COBOL 程序，大都是为了处理存储在 IBM 打孔卡片上的 80 列记录而编写的。为了尽量减少孔洞所占用的卡片空间，年份通常设计成 2 位而不是 4 位，随着时间的推移，这个设计的缺陷逐渐显露出来，最终导致在 2000 年出现了著名的"千年虫问题"（millennium bug）。

在 20 世纪 60 年代中期，为了配合 System/360 项目的开发，IBM 同时开发了程序设计语言 PL/ I（ I 是罗马数字中的 1，因此 PL/ I 的含义是：Programming Language Number One）。PL/ I 的设计者们想要使其融合 ALGOL 的块结构，FORTRAN 语言的数学函数功能以及 COBOL 处理记录和报表的能力，但该语言却远没有达到 FORTRAN 和 COBOL 那样广泛的使用程度。

虽然 FORTRAN，ALGOL，COBOL 以及 PL/ I 都可以应用于家用计算机，但它们对于小型计算机的影响远没有 BASIC 语言那么深远。

BASIC（Beginner's All-purpose Symbolic Instruction Code）由达特茅斯（Dartmouth）大学数学系的约翰·克莫尼（John Kemeny）和托马斯·克鲁兹（Thomas Kurtz）在 1964 年开发，该语言最初是为达特茅斯分时系统而设计的。达特茅斯大学的学生并非数学或工程专业，因此他们不应该为打孔卡片和复杂的程序语法花费太多精力，他们要做的只是端坐于计算机终端前，在数字后面输入一些 BASIC 语句来完成编程。BASIC 语句前的数字用来指明该语句在程序中的次序。前面没有数字的语句是系统命令，如 SAVE（将 BASIC 程序保存至磁盘），LIST（按顺序显示行）以及 RUN（编译并运行程序）。BASIC 手册的第一版中的第一个程序是这样的：

```
10 LET X = (7 + 8) / 3
20 PRINT X
30 END
```

与 ALGOL 语言不同，BASIC 不要求程序员指定变量的存储类型，究竟一个变量是保存为整型还是浮点型并不需要程序员担心，大部分数默认都是以浮点数格式存储的。

很多 BASIC 的后续版本都是解释型（interpreter）而不是编译型（compiler）。如前所述，编译器读取源文件并生成一个可执行文件；而解释器却采取边读边执行的方式，不会产生新的文件。解释器比编译器的原理简单一些，因此更容易编写，但其运行程序的速度要比后者要慢。BASIC 语言应用于家用计算机的时间较晚，1975 年，比尔·盖茨（Bill Gates，生于 1955 年）和其好友保罗·艾伦（Paul Allen，生于 1953 年）为 Altair 8800 编写了 BASIC 解释器，这一事件可以视为 BASIC 在此领域的开端，同一年他们创建了微软公司（Microsoft Corporation）。

Pascal 程序设计语言继承了 ALGOL 的大部分结构，同时还继承了 COBOL 的记录处

理功能，它由瑞士计算机科学教授尼尔莱斯·沃思（Niklaus Wirth，生于1934年）在20世纪60年代末开发完成。IBM PC的程序员对Pascal非常青睐，而备受欢迎Pascal版本却是大名鼎鼎的Turbo Pascal。1983年，宝兰公司（Borland International）发布了Turbo Pascal，当时的售价是49.95美元。Turbo Pascal由一名叫安德斯·海尔斯伯格（Anders Hejlsberg，生于1960年）的丹麦大学生开发，它提供了完整的集成化开发环境（integrated development environment）。程序的文本编辑器和编译器集成在一起，这样就方便了程序的调试和运行，大大加快了程序开发速度。集成化开发环境以前主要用于大型计算机，Turbo Pascal实现了在小型计算机上的突破。

Pascal对Ada的影响也非常大。Ada是为美国国防部开发应用的一种语言，它以奥古斯塔·艾达·拜伦（Augusta Ada Byron）命名。在第18章曾提到过，奥古斯塔·艾达·拜伦是查尔斯·巴贝芝的解析机发展历程的记录者。

接下来就是C，一种深受喜爱的程序设计语言。C语言主要是由贝尔电话实验室的丹尼斯·M·里奇（Dennis M.Ritchie）开发的，从1969年开始设计并于1973年开发完成。人们常常对为什么以C来命名该语言感兴趣，答案其实很简单，它是一种早期的程序设计语言B的后继者。B是BCPL（Basic CPL）语言的一种精简版本，而BCPL来源于CPL（Combined Programming Language）。

如第22章所述，UNIX操作系统在设计的过程中充分考虑到了可移植性。当时的许多操作系统都是基于某种处理器的，并且使用汇编语言编写，基本上没有可移植性可言。1973年，UNIX采用C语言编写（更准确地说，应该是重写）成功，从此以后UNIX操作系统和C语言就变得密不可分了。

C是一种风格非常简洁的语言。例如，ALGOL和Pascal使用关键字begin和end来界定程序块，而在C中这两个单词被一对大括号“{}”取代。下面给出一个例子，程序员常常会把一个常量和一个变量相加，比如：

```
i = i + 5;
```

在C程序中，你可以将上面的语句简写为：

```
i += 5;
```

如果只需要把变量加1（即增量），则该语句还可以精简成下面这样：

```
i++;
```

在 16 位或 32 位微处理器中，i++这种语句仅需要一条机器码指令就可以执行。

在本章的前面曾讲过，很多高级语言都不支持移位操作和按位布尔运算操作，而许多处理器其实支持这类操作，C 语言打破了这种局限，它广泛地支持这类运算。除此之外，C 语言的另一重要特征是对于指针（pointer）的支持，指针本质是数字化描述的内存地址。C 语言中的很多操作与通用处理器的指令非常相似，因此 C 也被称为高级汇编语言（high-level assembly language）。与类 ALGOL 语言相比，C 的操作集与通用处理器的指令集接近程度更高，或者说远胜过它们。

但是，所有的类 ALGOL 语言——即大多数常用程序设计语言——其设计模式都是基于冯·诺依曼计算机体系的。设计一种非冯·诺依曼体系的程序设计语言并非易事，而让人们接受并使用这种语言则更加困难。LISP（List Processing）是一种非冯·诺依曼体系程序设计语言，它主要应用于人工智能领域，由约翰·麦卡锡（John McCarthy）在 20 世纪 50 年代末期开发完成。APL（A Programming Language）是另一种全新的语言，与 LISP 完全不同，它同样完成于 20 世纪 50 年代末期，由肯尼斯·艾佛森（Kenneth Iverson）开发。APL 的特殊之处在于，它使用一个特殊的符号集，利用其中的符号可以一次性对整个数组里的数字完成操作。

类 ALGOL 语言一直在程序语言领域占据着重要地位，而且近年来，此类语言在一些方面进行了改进，导致面向对象程序设计语言（object-oriented language）的产生。面向对象语言主要应用在图形化操作系统中，我们将会在下一章（也是最后一章）介绍这种操作系统。

25

图形化革命

对于《生活》（*life*）杂志的读者而言，1945 年 9 月 10 日这一天的杂志像以往一样，有很多习以为常的文章和照片：第二次世界大战结束的相关新闻；讲述舞蹈家瓦斯拉夫·尼金斯基（Vaslav Nijinsky）在维也纳生活的点点滴滴；主题为美国汽车工人的图片新闻。但同时，在这一期的杂志中还有些不寻常的内容：万尼瓦尔·布什（Vannevar Bush 1890–1974）发表了一篇关于未来科学大胆猜想的文章。万·布什（人们常这样称呼他）的发明与贡献对计算机历史产生了深远的影响——其中最著名的就是他设计开发具有划时代意义的模拟计算机——微分分析器（The Differential Analyzer）——1927～1931 年，当时万·布什在担任麻省理工学院（以下简称为 MIT）工程学教授期间发明了这个机器。这篇文章在杂志上发表的时候，也就是 1945 年，布什所担任的职位是科学研究及开发办公室（Office of Scientific Research and Development，OSRD）的主任，负责美国战时科研活动的协调工作，其中就包括了曼哈顿计划（Manhattan Project）。

万·布什将自己两个月前在《大西洋月刊》（*The Atlantic Monthly*）上发表的一篇文章，通过浓缩精简，重新发表在《生活》杂志上，并将这篇文章最终取名为《思维之际》（*As We May Think*），文中描述了一种未来的发明，这项发明可以帮助科学家和研究人员更轻松地处理日益增多的技术期刊及文章。布什提出可以利用微缩胶片作为解决方案，同时他构想出了一种叫做麦克斯储存器（Memex，又名记忆扩展器）的设备，它可以对书籍、

文章、录音和图片进行保存。麦克斯储存器还有一项重要的功能，那就是它可以让用户根据某个主题在所有的素材之间建立起关联，这些关联的基本来源就是我们人类的思维。他还大胆预言一种新的职业群体，他们的工作就是在繁杂的信息载体之间提炼并建立起可靠的关联。

在 20 世纪的那个年代，讲述辉煌未来的文章屡见不鲜，但《思维之际》这篇文章却异常耀眼。它所讲述的不是可以替代我们去做家务劳动的设备，也不是关于未来运输方式或智能机器人的故事，这个故事的主角是信息（Information），故事的主线是如何利用新技术成功的处理信息。

回顾历史，从第一台继电器计算器出现到现在为止，65 年过去了，计算机的体积越来越小，处理速度越来越快，价格也越来越便宜。这一趋势极大地改变了计算的原始属性。当计算机价格变得很便宜，可以实现人手一台；当计算机体积越小、处理速度越快，软件就能发挥更大的作用，而机器就可以承担越来越多的工作。

要充分利用日益增长的运算和处理能力，较好的一种方法就是不断改进计算机系统中的关键部位，最典型的就是用户界面（User Interface）——它可以看作人机交互的轴心。人与计算机是两种完全不同形式的"客观存在"，只可惜在人机交互的这个过程中，与其让计算机去适应人类的特性，远不如劝服人们进行调整以适应计算机的特性来得容易。

在计算机发展早期，交互式这个概念并没有它的实际意义。人们编程时更多使用的是开关和电缆，有一部分人使用的是打孔纸带或胶片。到了 20 世纪 50 到 60 年代（有些观点认为这一时间可以延续到 70 年代），计算机已经可以使用批处理（batch processing）进行编程：程序和数据被"分布"在打孔卡上，然后一次性录入到计算机内存。这些工作完成之后，再由程序对数据进行分析，得出结论，最后将结果打印在纸上。

最早的交互式计算机运用的是电传打字机。我们回忆一下前面讲过的达特茅斯（Dartmouth）时分操作系统（原型出现于 20 世纪 60 年代早期），这种系统支持多个电传打字机同时工作，而且互不影响。此类系统中，用户在打字机上输入一行，计算机会相应地输出一行或多行。通常，打字机和计算机之间的信息交流是由一串 ASCII 码（也有可能是其他字符集）来完成的，这些 ASCII 码大多由字符编码组成，当然还包括像回车、换行等一系列简单的控制字符编码。随着机器的运行，相应的事务也随着打印纸的旋转逐步推进。

阴极射线管（cathode-ray tube，CRT，这是 20 世纪 70 年代随处可见的设备）并不受这类限制。使用软件来协调整个屏幕显得更加灵活方便——这可以算得上是一种二维的信息平台。但是为了尽量保持操作系统显示输出的逻辑一致性，早期那些为小型计算机编写的软件都把 CRT 显示器看做“玻璃屏幕电传打字机”——所有内容都是一行一地显示的，当字符排到底端，屏幕被填满时，屏幕上的内容要整体向上翻滚。除了 CP/M（微处理机操作系统）中的所有工具软件之外，大部分 MS-DOS 下的工具软件都采用这种方法——它们都仿照电传打字机的工作方式来使用视频显示器。使用电传打字机这种工作原理的操作系统有很多，或许 UNIX 才算是最典型的原型操作系统之一，它还一直保留着这种“传统工艺”。

不巧的是，ASCII 码字符集不完全适用于阴极射线管的工作方式。在最原始的 ASCII 码设计中，编码 1Bh 被标识为 Escape，它的主要作用是帮助字符集进行扩充。在 1979 年，美国国家标准协会（American National Standards Institute，ANSI）发布了一项题为“ASCII 码使用的附加控制（*Additional Controls for Use with American National Standard Code for Information Interchange*）”的标准。该标准发布的初衷是为了“适应二维字符-图像设备输入/输出控制中迫在眉睫的相关需求，其中包括阴极射线管和打印机之间的交互终端……”

其实 Escape 的编码 1Bh 只占据一个字节，且它的含义是唯一的。Escape 如果作为一串序列的前缀字符，那么这串字符序列的含义也随之改变。比如下面这串序列：

```
1Bh 5Bh 32h 4Ah
```

可以看出 Escape 编码随后紧跟的是字符“[”“2”“J”的 ASCII 码，现在这一串字符的含义为“清屏”然后移动光标至左上角。这种定义在电传打字机上是不可能出现的。下面这串序列：

```
1Bh 5Bh 35h 3Bh 32h 39h 48h
```

即 Escape 编码随后紧跟的是字符“[”、“5”、“;”、“2”、“9”、“H”，这串字符的作用是把光标移到第 5 行的第 29 列。

键盘和 CRT 一起对远程计算机传输来的 ASCII 码（可能还包括 Escape 字符序列）做出响应，这种设备我们称之为哑终端（dumb terminal）。哑终端相对于电传打字机速度要更快，从某种程度来讲也更灵活，但从速度的提高程度上来讲，并不足以引领用户界面

的革新。真正的革新出现在 20 世纪 70 年代小型计算机中——它类似于第 21 章我们构建的假想计算机，配备了“视频显示存储器”，并作为微处理器地址空间的组成部分。

第一个预示着家用计算机将与它的孪生兄弟——体积庞大、价格昂贵的大型机划分界限的标志性的事件是 VisiCalc 的使用。VisiCalc 由丹·布莱克林（Dan Bricklin，生于 1951 年）和鲍勃·弗兰克斯顿（Bob Frankston，生于 1949 年）设计并编程实现，而这套系统于 1979 年引入苹果 II 型电脑（Apple II）中。VisiCalc 通过屏幕将一个二维电子数据表呈现给用户。在 VisiCalc 出现之前，数据表就是一张划分好了行、列的纸，主要用于一系列计算。VisiCalc 用视频显示器将纸质材料取而代之，通过这种方式，用户可以在数据表中随处游走，在相应位置输入数据、公式，并在修改后对结果进行重新计算，为用户提供了更多的自由。

令我们惊讶与无奈的是，VisiCalc 这款应用程序无法在大型机上运行。因为像 VisiCalc 这类程序需要以较快的速度不断更新屏幕，所以，它们直接将数据写入 Apple II 视频显示器所配备的 RAM 中。该 RAM 是微处理器地址空间的一部分。大型时分计算机以及哑终端之间的接口速度过慢，以至于电子报表程序无法使用。

计算机对键盘的响应速度越快，对视频显示器的更新速度越快，则人机交互就越频繁。在 IBM PC 刚刚推出的 10 年里（即 20 世纪 80 年代），几乎搭配的所有软件都是直接将输出的数据写入视频显示存储器的。当时 IBM 建立了一套硬件标准，其他硬件制造商参照这些标准去生产，这样软件制造商就可以绕过操作系统直接操控硬件，统一化的硬件标准确保了程序的正确运行（同时也杜绝了不能运行的情况）。如果所有同构的 PC 都拥有异构的视频显示器硬件接口，这种做法无异于将软件厂商推到了火坑里，因为做软件的同时还要关注硬件设计细节是不现实的。

IBM 早期 PC 配备的应用程序通常只有字符输出，很少有图形输出。使用文本输出大大加快了应用程序的运行速度。假设 PC 上配备一台第 21 章所描述的视频显示器，那么程序所要做的就是把字符相应的 ASCII 码写入内存，然后屏幕上就会显示出该字符。但是如果使用的是图形视频显示设备，那么相应的程序需要将 8 个或更多的字节写入到内存中，这样做的目的就是画出字符的外观并以图形的方式显示。

在计算机的发展史上，从字符显示到图形显示是一次伟大的变革，计算机在这次变革中迈出了重要的一步。然而，相对于显示文本和数字所采用的软硬件，图形化计算机

的软硬件发展十分缓慢。早在 1945 年，约翰·冯·诺伊曼（John von Neumann）就预见了一种类似示波器的显示器，它的最大特点是可以显示图像化信息。但直到 20 世纪 50 年代早期，MIT（当时得到了 IBM 资助）建立了林肯实验室，实验室的主要任务就是帮助美国空军开发一种适用于防空系统的计算机，这次项目使计算机的图形化成为现实。该项目被称为半自动地面防空系统，简称 SAGE（Semi-Automatic Ground Environment），项目的内容包括构建一个显示图形的屏幕，以此来帮助操作员分析海量数据。

早期的视频显示器，比如 SAGE 中使用的这一种显示器，与我们今天所使用的 PC 配套显示器不尽相同。我们日常所用的 PC 配套的纯平显示器属于光栅（raster）显示器。它的原理就像电视机，每一幅图像背后都是一行行的光栅线，这些光栅是电子枪（electron gun）发出光束迅速来回移动覆盖整个屏幕而形成的。我们可以把屏幕想象成一个巨大的矩形阵列，阵列的每个元素都是一个点，这些点称为像素（pixels）。在计算机内部，有一块专门供视频显示器使用的内存区域，屏幕上的每一个像素点由 1 个或多个比特表示。这些二进制数值不仅决定了像素点的亮度，还决定了它的颜色。

举例来讲，当今大多数计算机显示器的水平分辨率至少为 640 个像素值，垂直分辨率至少为 480 个像素，像素总和即两数之乘积：307,200。如果为每个像素赋予 1 比特内存空间，这时每个像素点只能有两种颜色，通常设置为黑、白两种颜色，比如可以设置让 0 代表黑色，1 代表白色。这种视频显示器需要占据 307,200 比特的内存，换算过来就是 38,400 字节。

由于要用到的颜色数目逐渐增加，为了表示这些颜色，每个像素所需要的比特越来越多，显示适配器需要配备的存储器容量也越来越大。比如我们想使像素点具备不同的灰度，那么可以提供一个字节的存储空间。在这种处理方式下，字节 00h 代表着黑色，FFh 代表着白色，两者之间的值代表着不同的灰度。

CRT 上的色彩空间由三个电子枪产生，每一个电子枪分别产生三原色中的一种，包括红色、绿色、蓝色（用放大镜来观察电视机或彩色计算机屏幕，你可以清楚地看到，每一幅图像都是利用许许多多不同的三原色组合显示出来的），红绿组合出黄色，红蓝组合出品红色，蓝绿组合是青色，三原色组合出白色。

在最简单的彩色显示适配器中，表示每个像素点需要 3 个比特。最直观的编码方式就是每一种原色对应编码中的 1 位。

比特	色彩
000	黑
001	蓝
010	绿
011	青
100	红
101	品红
110	黄
111	白

这种方案可能只适合简单的类似卡通画的图像。真实世界出现的几乎所有颜色都是由红、绿、蓝三原色的不同色阶（levels）组合而成的。如果为每个像素赋予 2 个字节的存储空间，这样一来，可以给每一个原色分配 5 位（1 位保留）存储空间，这种方法可以表示出红、绿、蓝三种颜色且每种颜色具备 32 种不同的色阶，这样算下来总共有 32,768 种不同的颜色。这种模式通常称做高彩色（high color）或数千种颜色（thousands of colors）。

我们下面尝试一下用 3 个字节来表示一个像素，三原色中的每一种各占一个字节。这种编码模式使红、绿、蓝各自呈现出 256 种不同的色阶，这样算下来共有 16,777,216 种不同的颜色，这种方案通常叫做全彩色（full color）或百万种颜色（millions of colors）。如果视频显示器的分辨率为 640×480，即水平 640 像素，垂直 480 像素，将像素点的数量乘以表示每个像素点需要的字节数可以得到，共需要 921,600 字节的存储容量，即将近 1MB。

每个像素所赋予的比特数有时也称做色深（color depth）或色彩分辨率（color resolution）。颜色数与单个像素被赋予的比特数的关系如下：

$$\text{颜色数} = 2^{\text{每个像素所赋予的比特数}}$$

如果视频适配卡配备的存储器容量有限，那么它的最大色深或色彩分辨率自然而然也受到约束。假设有一个配备了 1 MB 存储器的视频适配卡，在每个像素被赋予 3 个字节的情况下分辨率可以达到 640×480。如果想把分辨率提高到 800×600，存储器就不足以为每个像素赋予 3 个字节，必须缩减到用 2 个字节来表示一个像素。

虽然现在来看在显示器上使用光栅技术似乎是很自然的事情，但是在早期，这种做法并不可行，因为在当时看来，这种技术需要的存储器空间太大。在这种情况下 SAGE 视频显示器应运而生，它是一种矢量（vector）显示器，相比电视机，它更像一种示波器。

电子枪可以通过电驱动定位到显示器任何一个像素点上，之后可以直接画出直线或曲线。由于屏幕上的图像具有持久性，不会立即消失，这样就可以利用直线和曲线形成最基本的画面。

支持光笔（light pen）是SAGE计算机的一大特色，操作者使用光笔可以改变显示器上的图像。光笔这种设备很特殊，从外观上来看是一端连有电线的笔。如果使用与之配套的软件，计算机能够感知到光笔所指的屏幕位置，随即根据光笔的位移相应地改变图像。

光笔的工作原理是什么呢？如果是第一次看到它，即使是相关领域的技术专家，也会感到困惑。理解它的关键在于光笔并不发射（emit）光——它所做的是检测（detect）光。对于CRT（无论采用的是光栅还是向量显示技术），电子枪移动控制电路有两个最重要的功能，第一个功能是光笔一旦感知到电子枪射出的光，系统需要立即做出反应；第二个功能是系统在对其做出反应的过程中，需要确定出光笔指向的屏幕位置。

伊凡·苏泽兰（Van Sutherland，生于1938年）是最早预见到了计算机发展的一个全新领域，即交互式计算的人之一。在1963年，他演示了为SAGE计算机专门开发的名为“画板”（sketchpad）的程序。画板不仅可以将图像信息存放在存储器中，还可以把图像在屏幕上显示出来。你还可以使用光笔在显示器上画出图像并进行修改，与此同时，计算机会对光笔的轨迹一直进行跟踪。

还有一位早期交互式计算的预言家，那就是道格拉斯·恩格尔巴特（Douglas Engelbart，生于1925年）。他曾阅读过1945年万·布什发表的文章《思维之际》，巧合的是，五年之后他开始致力于研究计算机界面显示的新方法，并为之奉献毕生精力。20世纪60年代中期，当恩格尔巴特在斯坦福研究所（Stanford Research Institute）工作时，他重新思考并设计了输入设备，提出了用五股（five-pronged）键盘作为指令输入设备（这个设备并未普及），另外还提出了一种配备轮子和按钮的设备，它的名字就是鼠标（mouse）。鼠标现在已经在全世界被广泛接受，它可以用来移动屏幕内的指针，还可以选择屏幕上出现的对象。

许多在早期热衷于交互式图形计算的科学家（但这里并不包括恩格尔巴特），他们不约而同地聚集在了施乐（Xerox）公司，幸运的是，此时的光栅显示器已经变得经济实用。施乐公司在1970年建立了帕洛阿尔托研究中心（Palo Alto Research Center，PARC），中

心的主要任务之一就是协助产品开发，以此来加快公司迈入计算机产业的步伐。PARC 中最著名的预言家应该算是阿伦·凯（Alan Kay，生于 1940 年），14 岁那年，阿伦·凯在一篇罗伯特·海因莱因 （Robert Heinlein） 撰写的故事中，读到了万·布什提出的微缩胶片图书馆，阿伦·凯因此而深受启发，不久他构想了一种名为“Dynabook”的便携式计算机。

PARC 着手的第一个大的工程是阿尔托（Alto），它的设计和制造完成于 1972-1973 年。从那个年代的标准去看，它是一个令人眼前一亮的产品。它采用落地式系统单元，配备 16 位处理器、2 个 3 MB 的磁盘驱动器、128 KB 的内存（最多可扩充到 512 KB），还包括一个三按钮的鼠标。在 Alto 开发的时候，16 位单芯片微处理器还未面世，所以它的处理器由将近 200 个集成电路组成。

Alto 有许多与众不同的地方，视频显示器是其中一个方面。屏幕的大小和形状就像一张纸——8 英寸宽，10 英寸高。它采用光栅成像技术，水平像素值为 606，垂直像素值为 808，算下来共有 489,648 个像素。其中每个像素占据 1 位存储空间，即每个像素取值只有两种：黑色或白色。视频显示的专用存储器容量为 64 KB，占用处理器的地址空间。

通过直接对视频显示存储器进行写操作，软件可以在屏幕上绘图或将不同字体、不同大小的文本显示在屏幕上。用户可以通过移动鼠标，在屏幕上对指针进行定位，还可以与屏幕上的对象进行交互。视频显示器与电传打字机在很多方面不尽相同，电传打字机顺序响应用户输入并按行将程序输出，而视频显示器的屏幕可以看做二维空间上的高密度的信息阵列，它还可以作为直接的用户输入源。

20 世纪 70 年代晚期，Alto 所搭配的程序逐渐凸显出很多新奇有趣的特点。比如窗口中可以容纳多个程序并同时显示在屏幕上。Alto 的视频图像功能使得软件从文本的束缚中摆脱出来，使其可以更加真实地反映用户的想法。图形对象（Graphical objects，比如按钮、菜单，以及被称做图标的小图片）成为用户接口的一员。鼠标可以在多个窗口中进行选择、触发图形对象来执行程序功能。

软件的内涵就在于此，它的意义远不止仅有的用户接口，还包括与用户的亲密耦合。软件使得计算机所涵盖的应用领域变得更广，而不仅仅局限于简单的数字变换。软件之所以被设计出来，其最终目的是——引用道格拉斯·恩格尔巴特在 1963 发表的一篇著名论文的标题——《为了扩展人类的智慧》（*For the Augmentation of Man's Intellect*）。

PARC 在 Alto 这个项目的开发成果预示着图形用户界面(Graphic User Interface，GUI)登上了历史的舞台。施乐公司并没有将 Alto 推向市场(价格定位 3 万美元以上绰绰有余)。从 10 年之后的今天来看，当时的 Alto 应该被包装成一种成功的消费产品并推向市场。

1979 年，斯蒂夫·乔布斯（Steve Jobs）带领苹果公司代表团对 PARC 进行了访问，在那里的所见所闻给他们留下了深刻的印象。而他们却花费了三年多的时间才推出具有图形界面的计算机，这就是在 1983 年 1 月推出的苹果莉萨（Apple Lisa），可惜这套系统在当时并不被看好。而一年以后推出的麦金托什机（Macintosh）却大获成功。

最原始的 Macintosh 机配备有 Motorola 68000 微处理器、64 KB 的只读存储器、128 KB 的随机访问存储器、一个 3.5 英寸的磁盘驱动器（存储容量为 400 KB)、一个键盘、一个鼠标和一个视频显示器，显示器水平像素为 512，垂直像素为 342（仅为 9 英寸的 CRT 对角线长度)，像素总量为 175,104 个。每个像素赋予 1 位内存，只能显示黑白两色，这种配置约占 22 KB 的视频显示存储器。

最原始 Macintosh 机硬件方面很精巧，但是可更新能力很差。1984 年 Macintosh 操作系统的诞生对于 Mac（即 Macintosh 机）意义非凡，它的出现使得 Mac 变得与众不同，当时我们把这样一个操作系统称为系统软件（System Software)，它就是现在著名的苹果操作系统（Mac OS)。

基于文本的单用户操作系统，如 CP/M 或 MS-DOS，体积很小但是不支持扩展的应用程序接口（API)。关于这点在第 22 章进行过解释，在这些基于文本的操作系统中，没有为访问文件系统的应用程序提供一种渠道。Mac OS 这种图形化操作系统所占的空间比前面提到的这两种要大得多，其中包含了上百个 API 函数，每一个函数都用其功能来命名。

MS-DOS 操作系统是基于文本的，如果要在屏幕上以电传打字机方式将文本显示出来，使用几个简单的 API 函数即可，但对于 Mac OS 这种基于图形的操作系统，必须提供一种在屏幕上显示图像的途径，程序通过这条途径对图像进行显示。从理论上来讲，一个 API 函数完全可以胜任这项任务，函数的功能就是设置某个水平和垂直坐标下的像素的颜色。但在实际应用中，这种方法效率较低以至于严重影响到了图像显示的速度。

在这种需求下，如果操作系统可以提供一整套图形编程系统，那么其意义是重大的，这样的操作系统必须包含如下 API 函数：画线、画矩形、画椭圆（包括圆）以及画出文

本。其中，线条可以是实线，可以是虚线，还可以是点线；矩形和椭圆可以具备不同的填充模式；字符可以具备不同字体和大小，还可以具备不同特效，如加粗和下画线等。图形编程系统负责规划如何将各式各样的图形对象以点阵集合的形式表示在显示器上。

如果程序在图形操作系统下运行，那么它们在显示器或打印机上画图这一过程中，使用的是一套完全相同的 API。正因为如此，字处理程序在屏幕上显示出的文档，与打印出来而得到的纸质文档，看上去非常相似。这种特点称为“所见即所得”（简写为 WYSIWYG）。这是喜剧演员弗雷普·威尔森（Flip Wilson）在扮演杰拉尔丁（Geraldine）角色中的一句话，这句话也成了计算机领域的一个经典口号。

图形用户界面对用户而言是极具吸引力的，其中一个重要原因就是不同的应用程序使用着大致相同的工作原理，并且影响着用户的使用经验。这样一来操作系统就承担起了支持 API 函数的重任，而应用程序就可以利用这些 API 函数去实现用户界面的不同组件，如按钮和菜单等。GUI 不仅是一种看上去简洁友好的用户环境，对于程序员而言，它还是一种重要的开发环境。程序员在开发新一代用户界面的时候可以不用从底层开始重新编写。

其实早在 Macintosh 问世之前，一些公司已经开始着手为 IBM PC 及其兼容机创建图形操作系统。这两种工作有一个显著的不同：苹果公司的硬件和软件都是由苹果公司自己设计的，因此开发人员的工作更加轻松。Macintosh 系统软件只支持一种类型的磁盘驱动器、一种视频显示器，以及两种型号的打印机。而 IBM PC 的图形操作系统开发人员所面对的是许多不同的硬件，操作系统与不同的硬件之间需要同时兼容。

还有一点，虽然 IBM PC 问世的时间（1981 年）较早，但 MS-DOS 应用程序已经在多数人心中根深蒂固，人们不愿意放弃它们。因此 PC 的图形操作系统必须具备一个重要的特性，那就是新的操作系统应该可以直接兼容 MS-DOS 应用程序，就好像 MS-DOS 应用程序是为新的操作系统专门设计的（Macintosh 不兼容 Apple II 系列软件，因为它们的微处理器型号不同）。

在 1985 年，迪吉多科研公司（Digital Research，CP/M 的后续公司）推出了图形环境管理器（Graphical Environment Manager，GEM）；VisiCorp（推出 Visilalc 软件的公司）推出了 VisiOn；与此同时微软公司发布了 Windows 1.0 版本，作为一匹黑马，当时它也被很多人认为将会成为“视窗争夺战”的胜利者。然而直到 1990 年 3 月 Windows 3.0 发布，

Windows 才真正受到大众瞩目。星星之火从那时开始燎原。在本书出版的 2000 年，约 90% 的小型计算机上使用的都是 Windows 操作系统。除了外观上的不同，Macintosh 和 Windows 这两种操作系统所包含的 API 也有着天壤之别。

从原理上来分析，除了图形显示器，图形操作系统与文本操作系统相比，对硬件支持的要求并没有太多不同。从理论上来讲甚至硬盘驱动器都可以算是多余的：比如最初的 Macintosh 没有配备，Windows 1.0 也不需要。虽然大家都认为使用鼠标操作会更加方便，但其实 Windows 1.0 可以不需要鼠标。

有一点很容易想到，随着微处理器速度越来越快，内存和外存的容量越来越大，图形用户界面将更加深入人心。图形操作系统将会支持越来越多的特性，它们所占的存储空间也将越来越大。2000 年左右的主流图形操作系统通常需要 200 MB 的硬盘空间和 32 MB 以上的内存。

在图形操作系统中，应用程序几乎都不使用汇编语言来开发。就拿早期的几款操作系统来看，Pascal 是 Macintosh 下的主流开发语言。在 Windows 操作系统中，C 语言一统江湖。还有一个不得不提到的例子，那就是 PARC 向我们展示的一种全新的方法。大概从 1972 年开始，PARC 的研究员着手开始研发一种名为 Smalltalk 的语言，这种语言嵌入了面向对象程序设计思想（Object-Oriented Programming），也就是今天的 OOP。

从传统意义上来讲，高级程序设计语言会自然而然地区分出代码（比如以 set、for、if 这样的关键词开头的语句）和数据，即变量所代表的数字。这种区分毫无疑问来自于冯·诺依曼计算机的体系结构。在这样一种体系结构中，只有两种元素，一种是机器码，一种是机器码所操作的数据。

在面向对象的程序设计中，和冯·诺依曼计算机的体系结构所不同的是，对象（object）实际上是代码和数据的组合。在对象内部，与其相关联的代码决定了数据存在的意义，要理解数据的存储方式首先需要理解代码。对象如果需要与其他对象通信，则通过发送或接收消息（message）来实现这一过程，比如一个对象可以通过给另一个对象发送指令来获得相应信息。

在图形操作系统的应用程序开发过程中，面向对象语言可以算得上是一种很不错的工具，因为编程人员处理屏幕上的对象（如窗口和按钮等）的过程就是用户感知屏幕元

素的过程。举例来讲，假设按钮是面向对象语言中的一个对象。屏幕上的按钮具备一定尺寸和位置，按钮上可以显示文本或小图标，这些都可以抽象成为与对象关联的数据。如果用户通过键盘或鼠标按下按钮，系统就会向按钮对象发送一个表示其被触发的消息，该按钮对象收到消息后就会调用与自身关联的代码进行响应。

小型计算机上最流行的面向对象语言是一种对传统的类似于 ALGOL 语言的扩展，C 和 Pascal 就属于此类。由 C 扩展的面向对象语言就是赫赫有名的 C++（我们可以回忆一下，两个加号放在一起等价于 C 语言中的自增操作）。C++的核心思想大部分来自于贝尔电话实验室（Bell Telephone Laboratories）的贾尼·斯特劳斯特卢普（Bjarne Stroustrup，生于 1950 年），最开始 C++是作为一种转换程序，它可以把编写的程序转换成 C 程序（但是转换出的C程序即难看又难以理解）。转换完成之后的C程序可以像普通程序一样编译。

其实，面向对象语言能做到的，传统语言也能做到。但是编程终究是人类发明的一种解决问题的活动，面向对象语言使得编程人员多了一种可选的解决方案，这种解决方案具备更加优越的组织结构。如果你想——虽然困难重重——面向对象语言编写的一种程序，并使其在 Macintosh 和 Windows 上都可以编译后运行，这是完全可以做到的。此类程序并不直接引用 API，使用的是被称为 API 函数的对象。Macintosh 和 Windows 使用两种不同的对象定义来编译程序。

许多在小型计算机上工作的编程人员已经逐渐不用命令行编译程序，而是使用集成开发环境（Integrated Development Environment，IDE）。这个环境里集成了所有需要的工具，而环境本身可以像其他图形应用程序一样运行，这样一来就大大简化了程序开发任务。还有一种称做可视化编程（Visual Programming）的技术被程序开发人员广泛利用，按钮及其他组件可以通过鼠标拖曳进行“排版”，从而达到在窗口交互设计的目的。

在第 22 章中我们一起讨论过文本文件。为了方便人们阅读，这类文件仅由 ASCII 字符组成。我们回想一下使用基于文本的操作系统的那个年代，文本文件是应用程序之间进行交流的理想媒介。它的最大优点是可检索性——程序可以检索多个文本文件，然后确定它们中是否有文件包含某一字符串。但如果操作系统中有一种机制用来显示不同字体、大小，以及不同效果比如斜体、黑体和下画线，那么文本文件就不再适用了。很多字处理软件其实都会使用一种自己独有的二进制格式来存储文档。文本文件同样也不适用于图形信息。

但我们要清楚的是，与文本相关的信息（比如，字体及段落版式），都可以被编码，而且编码后并不影响其可读性。这种方案的关键是选用一个适当的转换字符来标识出这些信息。在 Microsoft 设计的富文本文件格式（rich text format，RTF）中，大括号“{”和“}”以及反斜杠“\”封装了文本的格式信息，RTF 也成为了应用程序间传递格式化文本的一种方法。

PostScript 作为一种文本格式，将这种概念发挥到了极致。PostScript 的设计者是 Adobe 系统的创始人之一——约翰·沃诺克（John Warnock，生于 1940 年）。PostScript 是一种通用的图形编程语言，在 2000 年时主要用在高端计算机的打印机上，用于显示字符或图形。

随着硬件性能逐渐提升，价格日渐便宜，图形显示与个人计算环境的融合已是大势所趋。微处理器的处理速度越来越快，存储器价格越来越低廉，视频显示器及打印机分辨率不断增加，而且支持的颜色数目也成千上万，这一切大大推动了计算机图形界发展。

计算机图形也逐步产生了两种分支——本章的前面曾提到过这两个词，当时是为了区分图形视频显示器——这两个分支就是矢量和光栅。

矢量图形（vector graphics）在一些算法的帮助下，利用直线、曲线及填充区域生成图形。这也正是计算机辅助设计（Computer-Assisted Drawing，CAD）所应用的领域。矢量图形在工程和体系结构设计中有着十分重要的作用。矢量图形一般转化为图元文件（metafile）格式以存放到文件中。图元文件是由生成矢量图形的一系列绘制命令的集合组成的，这些命令通常已经被编码为二进制形式。

矢量图形的主要工具就是直线、曲线及填充区域。如果你想设计桥梁，使用矢量图形来实现将很简单，但如果要显示桥梁的实际结构，矢量图形就显得无能为力了。对于现实世界里的一副桥梁的整体结构图，用矢量图形来表示将会很复杂，而且困难重重。

光栅图形（也称做位图），就是为了解决这一问题应运而生的。位图（bitmap）将图像以矩阵阵列的形式进行编码，阵列中的一个单位对应着输出设备上的一个像素点。就像视频显示器一样，位图是一种空间上的概念（可以称其具有分辨率），其图像的宽度和高度都以像素为单位来表示。位图也具备色深（也可叫做颜色分辨率/颜色深度）的概念，色深是指每一个像素被赋予的比特数。位图中每个像素被赋予的比特数相同。

尽管位图从表现形式上看是二维的，但其本身的存储形式却是一串连续的字节——通

常从最顶端1行像素开始，紧跟着的是第2行、第3行，等等。

有些位图产生于图形操作系统设计的绘制程序，它们都是由某个操作者利用这些程序“手工绘制”出来的，还有一些位图是由计算机代码通过某种算法产生的。如今很多现实中的场景都利用位图来表示（比如数码照片），要把现实世界的图像输入到计算机中，可以借助一些不同的硬件，这类设备一般统称为电荷耦合器（charge-coupled device，CCD），它是一种在光照下会起电的半导体器件。每个像素都需要一个CCD单元来进行采样。

这些设备中最原始的可以算是扫描仪（scanner）了，其原理和影印机类似，都是利用一行CCD扫过需要复印的图像的表面，比如照片。由于光感度不同，不同区域CCD累积的电荷数也不同。扫描仪的配套软件把图像转换成位图存放在文件中。

视频摄像机利用二维CCD单元阵列捕捉图像。通常被捕捉到的图像存放在录像磁带上。但其实视频输出可以直接交给视频帧采集器（video frame grabber）去处理，它的工作原理是把模拟信号转换成像素值阵列。帧采集器可以收集通用的视频源，如录像机（Video Cassette Recorder，VCR）或激光影碟机（Laser Disc Player），甚至可以用于有线电视机机顶盒。

近几年，数码相机的价格已逐渐降低到了家庭用户可以承担的水平，它们从外观上来看和一般相机几乎一样。但数码相机并不使用胶片，而是使用CCD阵列来捕捉图像并直接将其转移到相机的存储器中，之后在适当的时候转储至计算机中。

图形操作系统通常可以将位图文件以某种格式进行存储。Macintosh 系统采用 Paint 格式，之所以叫这个名字是参考了创建这种格式的 MacPaint 程序（Macintosh 的 PICT 格式同时支持位图和矢量图形，而且是它们的首选格式）。Windows 里的默认的格式是 BMP，位图文件通常以它作为扩展名。

位图文件可能很大，如果有方法可以让它们变小一些那就再好不过了。这种需求催生了计算机科学中的数据压缩（Data Compression）这一全新领域。

假设我们正在处理一幅每个像素占 3 位的图像，这种图像在本章前面曾讲过。这张图片上出现的画面是一片天空、一栋房子和一块草坪。因此，图片中可能有大片的蓝色和绿色。很可能位图的最上面一行出现了72个蓝色像素。如果有一种方法可以表示蓝色

像素连续且重复了 72 次，那么通过这种方法表示的位图文件将会比原先的小很多。这样的压缩方法称为游程长度编码（Run-Length Encoding），即 RLE。

通常办公室的传真机采用的就是 RLE 压缩方法，压缩过程一般在传真机通过电话线传送图像之前。由于传真机展现出的图片都是黑白两色，没有灰度和彩色，所以通常像素值都会有很长串的白色区域，适合使用 RLE 压缩。

这十多年里风光无限的位图文件格式是图形交换格式（Graphics Interchange Format）即 GIF，由计算服务（CompuServe）公司于 1987 年开发。GIF 文件所采用的压缩技术称为 LZW，LZW 源自其三位创建者的名字：Lemplel、Ziv 和 Welch。LZW 比 RLE 更加强大，因为它所考虑的是像素值的模式（patterns），而 RLE 针对的是具有相同值的像素串。

RLE 和 LZW 都属无损（lossless）压缩技术范畴，因为可以从压缩数据中重新生成完整的初始文件。专业一点的说法是，压缩过程是可逆的（reversible）。可逆压缩方法并不适用于所有类型的文件，这点不难证明。在某些情况下，采用这些方法“压缩”后的文件比初始文件还要大！

近几年来看，有损（lossy）压缩技术大行其道。有损失的压缩是不可逆的，这是由于部分原始数据在压缩过程中被丢弃了。有损压缩技术不应该用于电子报表或文字处理文档，因为在这些重要文档里面少一个数字或者字母都会“失之毫厘，谬以千里”。但对于压缩图像，这些损失还是可以接受的，部分数据的损失不会使图片的整体效果有太大的变化。这就是为什么有损压缩技术的思想起源于心理视觉的原因，心理视觉领域所探究的是人的视觉，并根据心理因素确定人们所看到的景象中哪些比较重要而哪些不重要。

在 JPEG 中，人们使用了一系列具有重大意义的位图有损压缩技术，。JPEG 代表的是联合图像专家组（Joint Photography Experts Group），它涵盖了几种压缩技术，其中一些是无损的，另一些是有损的。

把图元文件转换成位图文件的方法很简单。这是由于视频显示存储器与位图在概念上保持一致。如果程序能把图元文件画在视频显示存储器中，则它也能在位图上画出图元文件。

但从位图文件到图元文件的转换却不那么容易，如果位图文件过于复杂甚至会导致无法转换。为了解决此类问题，人们发明了一项技术，那就是光学字符识别（Optical

Character Recognition），或简称为 OCR。对于位图上出现的一些字符（来自于传真机，或来自于页面扫描），如果需要转换成 ASCII 码，那么 OCR 就会派上用场。OCR 软件会对比特流进行模式识别，然后确定其代表的字符。由于算法的复杂性，OCR 软件并不能保证百分之百准确。虽然不很准确，但是 OCR 软件一直尝试将手写的字符也转换成 ASCII 码字符。

位图和图元文件都是数字化的可视信息。同理，音频信息也能转换成比特和字节。

随着 1983 年激光唱片（compact disc）的问世，数字化音响掀起了一轮消费狂潮，它同时也成为了最成功的电子消费品案例。CD 是由飞利浦和索尼公司联合开发出的产品，一张直径为 12 cm 的 CD 可存储 74 分钟的数字化声音。而这个 74 分钟的时长因为贝多芬的第九交响曲（Beethoven's Ninth Symphony）刚好可以存储在一张 CD 上。

CD 中声音信息采用的编码技术被称为脉冲编码调制技术（Pulse Code Modulation），简称为 PCM。名字听起来有点故弄玄虚，但从概念上讲 PCM 其实是一种很简单的过程。

振动是声音之源。人们发出的声音来源于声带的振动，大号的声音也来自于振动，森林里的树倒下的声音归根结底也是振动，还有很多例子，它们的振动来源于空气分子的移动。空气在被推拉的过程中，有时压缩有时放松，有时向后有时向前，这个过程每秒钟可以达到成百上千次，最终使耳膜产生振动，从而我们能够听到声音。

声波可以被模拟，一个成功的例子就是 1877 年爱迪生发明的第一台电唱机，它利用锡箔圆桶表面上的隆起和凹陷部位录制和播放背景音乐。直到 CD 的出现，这种录制技术才发生了稍许变化，圆桶变成了光盘，锡箔变成了塑料。早期的电唱机是机械化的，但是后来，电子放大器被用来放大声音。声音可以通过麦克风上配备的可变电阻转换成电流，喇叭中的电磁铁又可以将电流转换为声音。

代表声音的电流与先前所讲过信号不同，本书之前讨论过的是在"连通——断开"之间跳变的 1/0 数字信号。声波的变化是连续的，因而产生电流的电压也是如此。在这里电流产生的目的是模拟（analog）声波。为了达到这个目的，我们使用了一种新设备，通常把它叫做模拟数字转换器（Analog To Digital Converter，ADC）——所有的功能集成在了一个芯片上——将模拟电压转换成二进制数表示。一定长度的数字信号将会被 ADC 所输出——通常长度为 8、12 或 16 个比特——它们组合在一起表明了电压的相对级别。如果

ADC 的转化长度为 12 比特，那么电压值的取值范围为 000h～FFFh，这样一来就可以区分出 4096 个不同的电压级别。

在一种名为脉冲编码调制（Pulse Code Modulation）的技术中，以电压形式表示的声波将以恒定的频率被转换成数值。而这些数值将以小孔的形式刻在光盘表面，通过这种方式，电压就以数值的形式被存储在 CD 上。要读取这些信息时，可以通过分析从 CD 表面反射的激光读取到所存储的数值。在播放声音的时候这些数值又被转换成电流，这一过程利用到了数字模拟转换器（Digital-To-Analog converter，即 DAC，DAC 还可以用在彩色图形板上，将像素值转换成模拟信号并传输至显示器）。

声波电压在恒定的频率下被转换成了数字，该频率被称为采样率（sampling rate）。1928 年，贝尔电话实验室的哈里·奈奎斯特（Harry Nyquist）通过证明得到了一个总要结论：采样频率应至少为被采样信号（即被记录和播放的信号）最大频率的两倍。人类可以听到的声音的频率范围通常为 20～20,000Hz。CD 使用的采样频率为每秒 44,100 次，比人类听觉范围最大频率的两倍还要大一些。

CD 中存储的声音的动态变化范围决定了每次采样的比特数，这个范围就是 CD 存储声音的最高与最低频率之差。这难免给人感觉有些复杂：电流通过不断变化来模拟声波，其达到的最高峰称为声波的振幅（amplitude）。我们感知到的声音强度是振幅的两倍。1 贝尔（bel，这个名称来源于 Alexander Graham Bell 最后一个单词的三个字母），代表着强度的 10 倍；1 分贝（decibel）代表着 1 贝尔的十分之一。1 分贝代表着人类所能感知到的声音最小强度变化。

如果每次采样大小为 16 比特，这样就能够表示 96 分贝的动态范围，这一范围的下限是刚好能听到的声音的阈值（低于这一值的声音是听不到的），而上限就是人们承受最大负荷声音的极限阈值。CD 光盘的每个采样点就是用 16 比特表示的。

综上所述，CD 光盘中每秒产生 44,100 个采样样本，每个样本占据 2 个字节。有时你还希望享受立体声效果，这样的话采样信息需要翻倍，则每秒总共需要 176,400 字节，算下来每分种需要 10,584,000 个字节（这是个庞大的数字，正因如此，20 世纪 80 年代前数字记录声音的方法并没有普及）。74 分钟的立体声 CD 需要字节数为 783,216,000。

相对于模拟声音，数字化声音的优点不言而喻。特别是在模拟声音被复制的时候（例

如把录音磁带转录成电唱片）难免会有些失真。而对于用数字形式表示的数字化声音而言，都可以实现无失真的转录或复制。细心观察的人会发现，过去在电话通话中，信号传输线路越长则声音越糟。这种情况已经一去不复返了，因为现在大部分电话系统都已经数字化了，即使你打跨国电话，听筒中的声音也像在对街对话一样清晰。

CD 可以存储声音，也可以存储数据。专门用来存储数据的 CD 统称为 CD-ROM（CD 只读存储器），通常 CD-ROM 最大存储容量约为 660 MB。如今许多计算机中都配备 CD 驱动器，而应用程序及游戏软件都可以存储在 CD-ROM 中。

声音、音乐、视频逐渐走入个人计算机是在大约 10 年前，当时统称为多媒体（multimedia），虽然这几年发展很迅速，但这个名字已经普遍使用开来，所以也就不需要新起一个特别的名称了。如今的家用计算机都配备声卡，声卡中包含一个 ADC，它可以将麦克风传输的模拟声音信号转储成为数字信号，此外还包括一个 DAC，它的作用是帮助扩音器播放录制的数字声音。声音还可以按照波形文件（waveform files）方式存放于磁盘。

在我们使用家用计算机录制和播放声音的时候，其实对声音质量要求并不高，一般不会苛求达到 CD 的效果，所以 Macintosh 和 Windows 这两种操作系统提供了较低的采样频率，尤其是 22,050 Hz、11,025 Hz 和 8,000 Hz 这三种频率，采样信息量也保持在较少的 8 位，并且使用了频度录制手段。声音录制时所占的存储容量也减少到了每秒 8000 字节，这样算下来每分钟约占 480,000 字节。

很多人在科幻电影及电视剧集中看到过这样的场景，未来的计算机正在用纯正的英语与用户进行交互。只要计算机配备了录制和播放数字化声音的硬件，那么实现这个目标不过是一个软件问题。

如果要使计算机能够用易于理解的单词和句子与人交谈，有很多种方法。一种方法是预先录制句子、短语、单词还有数字，然后将它们以文件的形式存储，之后使用多种形式将其糅合在一起。这种方法常见于电话公司的信息系统中，在这种环境下只需要有限的单词、数字及其组合，因此这种方式能适用。

有一种更加通用的声音合成方法，它把 ASCII 码字符转换成波形数据。例如在英语拼写方面，由于存在很多不一致性，软件系统需要使用一个词典或复杂的算法来确定单

词的准确发音。一个完整的单词由基本的音节（也叫做音素，phonemes）组成。一般情况下软件都需要做一些其他方面的调整。如果一个句子后紧跟着的是问号，则最后一个单词的发音频率必须相应提高。

语音识别（voice recognition）——把波形数据转换成 ASCII 码字符——这个是一个极其复杂的过程。其实许多人在理解口语化的方言时也有很多困难。在使用个人计算中的语音处理软件时，通常需要对软件进行样本训练，以便软件能尽量准确地转录某个人的话语。这中间涉及的问题已经大大超出了 ASCII 码文本转换的范畴，从本质上来讲是一个利用编程技术使得计算机“理解”人类语言的过程。这类问题正是人工智能（artificial Intelligence）所研究的领域。

当今计算机中的声卡还配备了小型电子音乐合成器，它能模仿 128 种不同的乐器和 47 种不同的打击乐器，这类设备被称做 MIDI 合成器。MIDI 即乐器数字接口（Musical Instrument Digital Interface），这项发明出现于在 20 世纪 80 年代早期，由一家电子音乐合成器制造协会推出，主要用于将电子乐器组合起来，并且连到计算机上。

MIDI 合成器的类型不同，它们合成乐器声音的方法也就不同，有一些方法的效果更加逼真。MIDI 合成器所展现出来的特性已经超越了 MIDI 所定义的范畴。但其实它的功能就是以演奏声音的方式来响应短消息序列——长度通常为 1、2 或 3 个字节。MIDI 消息本身就指明了所需要的乐器、将要演奏的音符，或正在演奏的音乐的休止符。

MIDI 文件不仅包含了一系列 MIDI 消息集合，还包括了时间信息。通常一个 MIDI 文件“麻雀虽小，五脏俱全”，包含了整套演奏信息，因此可以由计算机上的 MIDI 合成器完整地演奏。相比于包含同样一段音乐的波形文件，MIDI 文件通常要小得多。就文件的相对大小而言，如果把一个波形文件比作位图文件，则 MIDI 文件就好像矢量图元文件。MIDI 文件的缺点来源于 MIDI 合成器的异构性：同样一个 MIDI 文件可能在一个合成器上完美演奏，但在另一个合成器上可能不堪入耳。

数字化电影是多媒体的另一片天地。把一系列静止图像快速播放，可以达到电影和电视图像中出现的物体移动效果。我们把这期间出现的单个图像称为帧（frames）。电影的播放速率为 24 帧/秒，北美的电视节目为 30 帧/秒，世界上大部分地方的电视为 25 帧/秒。

计算机中的电影文件一般都是由一系列附带声音的位图组合而成。但是如果不经过压缩处理，一部电影文件中的数据量将会很大。假如电影中每一帧包含的像素大小是640×480，每个像素为24位真彩色，那么每一帧的大小就为921,600字节。如果播放速度为30帧/秒，则每秒需要的存储空间为27,648,000字节。照这样计算下去，每分钟需要的空间大小为1,658,880,000字节，一部两小时的电影需要199,065,600,000字节，大约200 GB。正因如此，我们的计算机上播放的电影经常很短、解析度很小，而且清晰度很差。

就像JEPG压缩技术可以用来减少静态图像所占的数据空间一样，MPEG压缩技术用于处理动态电影文件。MPEG全称是移动图像专家小组（Moving Pictures Expert Group）。动态图像压缩技术基于的是一种客观事实，即每一帧继承了前一帧的大部分信息，也就是说存在冗余信息。

针对不同的多媒体产品有不同的MPEG标准。MPEG-2用于高清晰度电视（high-definition television，HDTV）及数字影音光盘（digital video discs，DVD）。DVD也叫数字多用光盘（digital versatile discs），其物理大小与CD一样，但是DVD的两面都可以记录数据而且每一面有两层。在DVD中，视频信息可以压缩为原始大小的五十分之一，因此前面提到的一部两小时的电影将占据4 GB的空间，而且只占据其中一面的一层。所以一张具有两面四层的DVD盘容量可达到16 GB，容量达到了CD的25倍。如果不出预料，在未来的某一天，DVD将替代CD-ROM成为新的软件存储媒介。

随着CD-ROM和DVD-ROM的出现，这是否意味着万·布什预言的麦克斯储存器在今天成为了现实？最原始的麦克斯储存器设想中，使用的原材料是缩微胶片，但显然CD-ROM和DVD-ROM更合适担此重任。由于电子媒体易于检索，它们比物理媒体更具有优越性。可惜同时访问多个CD或DVD驱动器并不现实。我们的存储设备有一点与万·布什所提出的概念有所不同，那就是所有信息并不一定要触手可及，真正使它们达到信息共享的做法是计算机互连，这样做还可以更有效地利用存储空间。

对计算机进行公开性远程操作的第一人是乔治·史帝比兹（George Stibitz），也正是他，在20世纪30年代设计了贝尔实验室的继电器计算机。对继电器计算机的远程操作的演示地点在达特茅斯，时间是在1940年。

电话系统在线路上传输的是声音，而不是比特。在电话线路上传输比特需要先将其转换成声音，传输完之后在转换回比特。单一频率和振幅的连续声波（统称为载波，carrier）

无法表达完整清晰的信息。但如果对声波进行一些调整——说得专业一点就是，对声波进行调制（modulate）使其反映两种不同的状态——通过这种方式可以表示出 0 和 1。将比特与声波进行互相转换的设备被称做调制解调器（modem，它包括调制和解调两个功能）。调制解调器以串口（serial）形式工作，因为字节中的单个比特是一个接一个传输的，而不是一拥而上（打印机一般通过并行接口与计算机相连：整个字节由 8 根线并行传输）。

早期调制解调器采用了频移键控（frequency-shift keying，FSK）技术。假设调制解调器的处理速度为 300 bps，而且二进制数 0 被调制到 1070 Hz，而 1 被调制到 1270 Hz。每个字节被夹在一个起始位和一个停止位中间，所以每个字节其实占据了 10 位空间。在 300 bps 的传输速率下，每秒传输的字节数为 30 个。现在许多采用先进技术的新一代调制解调器速度超过了它的 100 倍。

早期家用计算机的狂热爱好者可以使用计算机和调制解调器建立电子公告牌系统（Bulletin Board System，BBS），其他计算机可以接入到这个系统中并下载（download）文件，这意味着文件从远程计算机中传输到了自己的计算机。这些概念发展广泛，甚至于扩展到了大型信息服务领域中，例如线上资料库服务（CompuServe）。在大多数环境下，通信中采用的一般都是 ASCII 码。

Internet 与这些早期的发明有本质上的差别，因为它是一种非中心化的系统。Internet 从本质上来讲是一组协议的集合，这些协议是计算机之间相互通信的保证。众多的协议中，最重要的当属 TCP/IP 协议，它包括了传输控制协议（Transmission Control Protocol，TCP）和网际协议（Internet Protocol，IP）。这种协议使得传输过程变得规则化，不再是简单地通过线路传输 ASCII 码字符，而由基于 TCP/IP 协议的传输系统把大的数据块分割成小的包（packets），之后在传输线路上（通常是在电话线上）独立发送，最后在传输线路的另一端将数据重新组装起来。

Internet 中最流行的是万维网（World Wide Web），而这一部分与图形联系最紧密。万维网采用了 HTTP 协议来支持其工作，HTTP（Hypertext Transfer Protocol）即超文本传输协议。几乎所有在 Web 页面上看到的数据都遵循一定格式，那就是 HTML（Hypertext Markup Language），即超文本标记语言。其中超文本（hypertext）这个单词用来描述相关链接信息，非常类似于万·布什预言的麦克斯储存器。HTML 文件可以包含指向其他 Web 页面的链接，这样可以轻松访问其他页面。

HTML 与本章前面讨论过的富文本格式（RTF）很相似，它们都含有带有格式信息的 ASCII 码文本。HTML 也可包含多种图片格式，例如：GIF 文件、PNG（portable network graphics，便携式网络图像格式）文件，以及 JFIF（JPEG 文件交换格式）等。大部分万维网浏览器都支持浏览 HTML 文件，这是文本格式的优势所在。易检索性也是 HTML 被定义成文本文件的另一个优点。虽然名字可能有些让人迷惑，但 HTML 与我们在第 19 章和第 24 章讲到的语言不同，它并不是真正的程序设计语言。Web 浏览器首先读取 HTML 文件，根据读取到的内容显示文本和图形并编排它们的格式。

当我们浏览某些 Web 页面并进行一些操作时，有一些特殊程序需要并发执行，程序中的代码可以运行在服务器端（Server，用来存储原始 Web 页面的计算机）或客户端（Client），我们自己的计算机就是客户端。服务器端责任重大，通常要完成一些重要的处理工作（例如解释客户端填写的在线表格），服务器端的工作可以通过公共网关接口（Common Gateway Interface，CGI）脚本来处理。而对于客户端，HTML 文件可以包含简单的程序设计语言，例如著名的 JavaScript。Web 浏览器可以对 Java Script 语句进行解释，就像解释 HTML 文本一样。

为什么 Web 站点不为我们的计算机提供一个可执行程序，如果这样的话问题一下子就解决了。要回答这个问题，我们首先要明确：我们的计算机是什么样的？如果使用的是 Macintosh 机，那么这个可执行程序需要包含 Power PC 机器码，而且需要引用 Mac OS 中的 API 函数；如果是一台 PC 兼容机，那么这个可执行程序需要包含 Intel Pentium 机器码，而且需要使用 Windows API 函数。但计算机及图形操作系统的种类繁杂，还有许多其他类型的计算机和图形操作系统。进一步来说，我们也不想毫无目的地下载可执行文件，它们很可能来自于非信任站点而且会带有恶意行为。

上述问题的答案就在 Sun 公司开发的 Java 语言中（请勿与 JavaScript 混淆）。Java 是一款成熟的面向对象程序设计语言，和 C++有些类似。前面章节中我们讨论过编译语言（compiled languages，可产生包含机器码的可执行文件的语言）和解释语言（不可产生可执行文件的语言）之间的区别，Java 是一种介于两者之间的语言。它需要经过编译，但编译的结果不是机器码，而是 Java 字节码（Java byte codes）。Java 字节码与机器码在结构上很相似，但 Java 字节码可以在一种虚拟的计算机下被解释，即 Java 虚拟机（Java Virtual Machine，JVM）上。被编译的 Java 程序产生 Java 字节码，之后计算机模拟 JVM 对其进

行解释。Java 程序的运行可以不受限于机器与图形操作系统的类型，所以它具有平台无关性（platform- independent）。

关于如何利用电流在线路上传输信号和信息，在讲述这一点时本书利用了大段章节，但其实有一种更加行之有效的方法，那就是利用光纤——一种由玻璃或聚合体制造的光导纤维，通过从不同角度对光进行反射达到光传输效果。光信号在光纤中的传输速率可以达到千兆赫兹——即每秒十亿个比特。

展望未来，在以后的家庭和办公室中，光子似乎要替代电子承担海量信息传输的重任。比起莫尔斯码中的一“点”一“划”，比起为了与一街之隔的好友深夜交流而绞尽脑汁想出的闪光灯，光子的速度无与伦比。